UCLouvain

École polytechnique de Louvain

Development of a programmable wireless receiver platform for Electrical Engineering education

Author: **Augustin Deper**
Supervisors: **David Bol, Martin Andraud**
Readers: **Jérôme Louveaux, Pol Maistriaux, Nicolas Brusselmans**
Academic year 2024–2025
Master [120] in Electro-mechanical Engineering

Abstract

The LimeSDR Mini reached end of life, leaving the LEPL2102/2103 courses of the EPL (Ecole Polytechnique de Louvain) without a practical way to access raw baseband signals for hands-on labs. This work fills that gap with a receiver platform that fits the existing workflow and remains easy to understand and maintain.

The design targets the sub-GHz band and keeps the established toolchain. A Microchip AT86RF215 front end on a REB215-XPRO-A board produces raw I/Q over an LVDS link. A DE10-Nano SoC FPGA captures the stream, performs basic integrity checks, and writes samples into system memory. The Hard Processor System forwards the data over UDP to a host computer where the flow is ingested by GNU Radio. The signal path is straightforward to follow from the antenna down to complex samples on the computer.

The contribution is a complete and documented replacement for the LimeSDR Mini in receive mode. It includes SPI bring-up of the radio, a stable LVDS capture in the FPGA fabric, a DMA pipeline to the HPS, and a host path that preserves packet order and flags gaps. The platform is reproducible, compatible with Quartus, and ready for labs focused on signal acquisition and early demodulation.

Acknowledgements

I would first like to express my sincere gratitude to Prof. David Bol and Prof. Martin Andraud for their guidance and supervision throughout this work. Their strategic advice and constructive feedback greatly shaped the work and helped keep it rigorous and focused.

I am also thankful to Nicolas Brusselmans for his availability and practical advice, and for his help at key moments of the project.

I am especially grateful to Pol Maistriaux for his outstanding practical assistance, including patient debugging, concrete suggestions, and the considerable time he invested to overcome technical obstacles. His involvement made a decisive difference.

Finally, I thank Souley Djandjandi for his support on the hardware side, in particular for facilitating equipment loans and for soldering the cables on the RF transceiver.

Additionally, this work used OpenAI's ChatGPT to assist with phrasing and language polishing.

Contents

Abstract i

Acknowledgements iii

List of Figures ix

List of Tables xi

Abbreviations xiii

1 Introduction 1

2 State of the Art and Preliminary Analysis 5

2.1 Fundamentals of Software Defined Radio (SDR) 5

2.1.1 General architecture of an SDR system 5

2.1.2 Digital modulation schemes: BFSK / GMSK modulation 9

2.1.3 Digital processing of I/Q signals 12

2.2 Analysis of the current platform (LimeSDR Mini) 14

2.2.1 General overview 14

2.2.2 Technical specifications and hardware architecture 15

2.3 Technological choices for the new platform 20

2.3.1 FPGA platform: DE0-Nano vs. DE10-Nano 20

2.3.2 Software tools: signal processing and visualization 22

2.3.3 Radio transceiver chip: selection criteria and comparison 23

2.3.4 RX Shield: a custom PCB? 26

2.3.5 Final system architecture 28

3 Platform Design and Development 31

3.1 Overall system architecture . . . 31
3.1.1 General overview . . . 31
3.2 Transceiver integration and configuration . . . 32
3.2.1 SPI configuration interface of the AT86RF215 . . . 32
3.2.2 LVDS DDR data transmission . . . 34
3.2.3 Configurable parameters and operational modes . . . 36
3.2.4 Pinout and Connections . . . 38
3.3 FPGA/HPS Embedded HDL/Software Development . . . 40
3.3.1 FPGA Architecture and PS/PL Partitioning . . . 40
3.3.2 SPI communication with the AT86RF215 . . . 43
3.3.3 LVDS Data Reception and Handling in the FPGA . . . 45
3.3.4 Pin assignments and constraints . . . 46
3.3.5 DMA Transfers Between the FPGA and the HPS . . . 47
3.3.6 HPS software and UDP transfers . . . 49
3.3.7 OS installation, configuration and bitstream loading . . . 51
3.4 Host computer software interface . . . 52
3.4.1 Host computer environment . . . 52
3.4.2 Network configuration . . . 52
3.4.3 Software architecture and UDP data-stream reception . . . 52
3.5 Installation of Software Dependencies . . . 54

4 Validation and Experimental Results . . . 57
4.1 Validation methodology and setup description . . . 57
4.2 Validation of data streams and interfaces . . . 59
4.2.1 SPI transfer integrity . . . 59
4.2.2 LVDS data integrity between transceiver and FPGA . . . 59
4.2.3 FPGA to HPS DMA data transfer reliability . . . 61
4.2.4 UDP data reception and integrity on PC . . . 65
4.3 Preliminary signal processing and demodulation tests . . . 67
4.3.1 Demodulation tests in GNU Radio . . . 67

5 Alignment with the objectives, limitations, and perspectives . . . 71
5.1 Alignment with the objectives . . . 71
5.2 Limitations of the current implementation . . . 73
5.3 Perspectives for future work . . . 74

6 Conclusion . . . 77

A AT86RF215 States and Registers . . . 83
A.1 Finite State Machine for the AT86RF215 Transceiver . . . 84
A.2 AT86RF215 Configuration Registers . . . 84

B FPGA Pin Assignments . . . 87

C Configuration and setup of the DE10-Nano Board . . . 89
C.1 Device-tree Modifications and Installation . . . 89
C.1.1 Required Modifications . . . 89
C.1.2 Recompiling and Installing the Device-tree . . . 91
C.1.3 Verification . . . 92
C.2 Bitstream Conversion and Upload Procedure . . . 92
C.2.1 Bitstream Conversion . . . 93
C.2.2 Uploading the Bitstream . . . 93
C.2.3 Notes . . . 93

D Memory-Mapped transactions between the HPS and the FPGA . . . 95
D.1 Accessing the FPGA peripherals from the HPS . . . 95
D.1.1 Local addresses of the FPGA peripherals . . . 95
D.1.2 Offset of the FPGA peripherals in the HPS memory map . . . 96
D.2 Accessing the DDR3 memory from the FPGA . . . 96
D.3 mSGDMA Register Maps . . . 97
D.3.1 Descriptor Slave Port . . . 97
D.3.2 Control/Status Registers (CSR) . . . 97
D.3.3 Response Port . . . 98

E DHCP server configuration on the host computer . . . 99

F Installation of Software Dependencies . . . 101
F.1 Installing the `gr-fsk` GNU Radio module . . . 101
F.1.1 Build and install . . . 101
F.1.2 Ensure the module is discoverable by GNU Radio . . . 102
F.1.3 Obtain and configure the evaluation flowgraph . . . 102
F.2 Software installation on the HPS . . . 103
F.2.1 Installing the control daemon . . . 103
F.2.2 Compiling the program . . . 104
F.2.3 Running the program . . . 105

List of Figures

2.1 Generic demodulation chain with illustrative signal snapshots for a BFSK example. . . . 8
2.2 GMSK vs. MSK modulation waveform. . . . 11
2.3 LimeSDR Mini overview. . . . 15
2.4 Simplified receive path in the LMS7002M (single channel). . . . 17
2.5 Comparison of the main package types for candidate transceivers. . . . 25
2.6 Simplified sub-1 GHz receive path in the **AT86RF215**. . . . 26
2.7 The ATREB215-XPRO-A evaluation board, integrating the AT86RF215 transceiver and all necessary support circuitry. . . . 28

3.1 Overall system architecture. . . . 32
3.2 SPI single *write* access; `SELN` remains low for the entire three-byte frame. 33
3.3 SPI single *read* access; `SELN` remains low for the entire three-byte frame. 34
3.4 Timing of the receive LVDS link in DDR mode. . . . 36
3.5 XPRO-A connector: schematic view and pin mapping. . . . 38
3.6 LVDS interface—SATA-style footprint on the REB215-XPRO-A PCB. 39
3.7 Flowchart of the software running on the HPS. . . . 41
3.8 Cyclone V block diagram: instantiated HDL IPs and interfaces with the HPS. . . . 42
3.9 `CS_fuser_edge` IP operation. . . . 44
3.10 Pin connections between DE10-Nano and the AT86RF215. . . . 48
3.11 GNU Radio simplified flowchart. . . . 55

4.1 SPI read transaction (SignalTAP capture). . . . 60
4.2 Interval between successive `word_valid` strobes extracted from SignalTAP. . . . 61
4.3 I and Q sample values with transmitter disabled (noise-only condition). 62

4.4 Comparison of the samples at the output of the LVDS_rx, at the mm_write bus of the DMA and read by the HPS from the DDR. . . . 63
4.5 Overload conditions in the DMA transfer process, as observed in the Quartus SignalTap logic analyzer. 64
4.6 Normal operation of the DMA transfer, as observed in the Quartus SignalTap logic analyzer. 64
4.7 Comparison of the samples sent by the HPS and received by the PC. . 66
4.8 QT Sink block interface. 69

A.1 Finite State Machine for the AT86RF215 Transceiver. 84

List of Tables

2.1 Core LMS7002M specifications relevant to this work. 16
2.2 Intel MAX 10 resources available on the LimeSDR-Mini. 18
2.3 High-level comparison of DE10-Nano and DE0-Nano development boards. 21
2.4 Resource utilization of the MAX 10 device 10M16SAU169C8G. . . 22
2.5 Key characteristics of the transceiver candidates. 24

3.1 Mechanical characteristics of the XPRO connector (REB215-XPRO-A board). 39

4.1 GNU Radio variables. 58
4.2 Hard-coded RF/filter register fields (currently not exposed to GNU Radio). 58

A.1 Configuration registers used in this project (sub-GHz, I/Q streaming mode). 85

B.1 FPGA pin assignments with corresponding GPIO indices. 88

D.1 Local address map of `msgdma_0` inside `soc_system` (relative to the interconnect window). 95
D.2 Base address ranges of the HPS-to-FPGA bridges (compute: base + Qsys offset). 96
D.3 Standard descriptor format written by the HPS to `descriptor_slave`. 97
D.4 mSGDMA CSR map (`csr` interface). 97
D.5 Status register bit definition. 98
D.6 mSGDMA response source port bit fields. 98

Abbreviations

ADC Analog-to-Digital Converter

AXI Advanced eXtensible Interface

BER Bit Error Rate

CFO Carrier Frequency Offset

CRC Cyclic Redundancy Check

CSR Control and Status Register

DDR Double Data Rate

DHCP Dynamic Host Configuration Protocol

DMA Direct Memory Access

ENOB Effective Number of Bits

EPL Ecole Polytechnique de Louvain

FIFO First In, First Out

FIR Finite Impulse Response

FPGA Field Programmable Gate Array

FSK Frequency Shift Keying

FSM Finite State Machine

GMSK Gaussian Minimum Shift Keying

GNU GNU's Not Unix

GPIO General Purpose Input/Output

HDL Hardware Description Language

HPS Hard Processor System

I/Q In-phase/Quadrature

IF Intermediate Frequency

IP Intellectual Property

LE Logic Element

LPF Low-Pass Filter

LSB Least Significant Bit

LVDS Low-Voltage Differential Signaling

MAP Maximum A Posteriori

MISO Master In Slave Out

MOSI Master Out Slave In

MSB Most Significant Bit

MSK Minimum Shift Keying

OCT On-Chip Termination

PL Programmable Logic

PLL Phase-Locked Loop

PSK Phase Shift Keying

QAM Quadrature Amplitude Modulation

RF Radio Frequency

SDR Software Defined Radio

SLVDS Scalable Low-Voltage Differential Signaling

SPI Serial Peripheral Interface

SSH Secure Shell

STO Symbol Timing Offset

TSP Transceiver Signal Processor

UDP User Datagram Protocol

1

Introduction

Forests are vital ecosystems for biodiversity, climate regulation, and human well-being, but they are increasingly threatened by disturbances such as wildfires, illegal logging, and insect infestations. Human activities—whether directly (e.g., logging, land conversion) or indirectly (e.g., climate change)—play a major role in these threats. For example, the Californian agency CAL FIRE reports that 95% of wildfires in California are caused by human activities, highlighting the significant impact of humans on forest disturbances [1].

Consequently, forest management and monitoring have become increasingly important in order to limit impacts on natural ecosystems. Early detection of forest disturbances is critical to maintaining forest health and sustainability, making efficient and reliable monitoring methods essential [2].

Each year, the students of the Electronic Master's program at the Université catholique de Louvain (UCLouvain) undertake a project in which they acquire, process, transmit, and finally classify audio data recorded in a forest. The goal is to analyze the acquired data to identify and classify different types of disturbances such as wildfires and illegal logging [3].

To carry out this project, students use a *Software-Defined Radio* (SDR), a radio communication system in which signal-processing tasks are performed by software rather than dedicated hardware. In a conventional radio, reception and transmission follow predefined standards, and the demodulation process that

extracts information from the radio signal is handled entirely by fixed hardware components, leaving the user without visibility or control. By contrast, an SDR provides access to the raw, un-demodulated signals, enabling full control over how they are processed in software. This flexibility is particularly valuable in an educational context, as it lets students experiment with different signal-processing techniques and algorithms.

For this educational activity, a LimeSDR Mini (v1) was initially adopted. The LimeSDR Mini combines a broadband RF transceiver with a field-programmable gate array (FPGA) that can be configured through the Quartus design environment, which is already familiar to students. Its plug-and-play nature allows rapid setup and immediate hands-on experience.

Despite its advantages, the LimeSDR Mini now presents several limitations in our educational context. Most importantly, it has been discontinued by the manufacturer, making it increasingly difficult to source new units or obtain support [4]. Its high level of integration complicates repairs and maintenance, and its complex configuration can prove challenging for newcomers.

Although a new version of the LimeSDR Mini (v2) has been released, it ships with a Lattice FPGA that is not compatible with the Quartus software used in the Electronics Master's program, and its purchase price is approximately €400. This per-unit cost is significant further motivates the development of an in-house solution.

Objectives of the work

The present work addresses these issues by designing a new receiver platform that is robust, easy to maintain, and well suited to educational needs. The specific objectives are:

1. Employ a widely used FPGA platform, already familiar to students and compatible with Quartus.
2. Fulfill the precise requirements of the ELEC Master's practical project.
3. Select an IQ transceiver module that is straightforward to implement and maintain.
4. Implement the complete data chain from antenna to host computer, passing through the radio transceiver and the FPGA.

Contributions

Within the scope of this work, a complete data chain has been designed, implemented, and validated. The radio transceiver is configured and sends I/Q samples to the FPGA, which forwards them to the host computer. The system enables full demodulation and end-to-end data processing. This modular architecture provides a robust foundation for future extensions in teaching and research.

The manuscript is organized as follows. Chapter 2 examines the current receiver setup and identifies its limitations. It also compares several FPGA platforms and IQ transceivers, leading to the selection of the most suitable components. Chapter 3 details the design and implementation of the new receiver chain, and Chapter 4 presents its validation through practical testing.

Ultimately, this work aims to deliver a student-friendly solution that is maintainable and easy to repair, enhancing future practical teaching and research activities within UCLouvain's Electrical Engineering Master's program.

2

State of the Art and Preliminary Analysis

2.1 Fundamentals of Software Defined Radio (SDR)

2.1.1 General architecture of an SDR system

Radio receivers are present in many everyday devices, such as smartphones (for Wi-Fi or cellular networks), speakers (for Bluetooth audio streaming or radio), and televisions (for digital broadcasting).

Depending on the application, these connections must meet different requirements in terms of bandwidth, latency, carrier frequency, and power consumption. This diversity leads to a variety of modulation technologies and hardware architectures. For example, Wi-Fi uses Phase-Shift Keying (PSK) or Quadrature Amplitude Modulation (QAM) at frequencies between 2.4 GHz and 5 GHz, while Bluetooth employs simpler schemes such as FSK and PSK to reduce demodulation complexity, as the required data rates are lower.

How is demodulation performed? What steps are required to extract data from a received signal?

1. **Antenna:** The antenna is the first component in the chain, receiving the radio signal and converting electromagnetic waves into electrical signals.
2. **RF Filtering and Amplification:** The received signal is typically mixed with noise and other signals. RF filters isolate the desired carrier frequency and bandwidth, removing unwanted components. Amplifiers (LNA / variable-gain stages) increase the strength of the filtered signal while preserving signal-to-noise ratio.
3. **Quadrature Downconversion:** The filtered RF signal is mixed with a locally generated oscillator (LO). Two mixers fed by LO signals in phase quadrature (0° and 90°) produce the in-phase (I) and quadrature (Q) baseband (or low-IF) components. This shifts the spectrum close to DC (zero-IF) or to an intermediate frequency (low-IF) where subsequent analog filtering and digitization are easier.
4. **Baseband Filtering and Sampling:** Low-pass (or channel-select) filters remove out-of-band noise, images, and mixer products. An analog-to-digital converter (ADC) then samples the I and Q signals at a rate meeting Nyquist criteria for the selected bandwidth. Optional decimation may follow.
5. **Demodulation / Digital Processing:** The complex (I/Q) sample stream undergoes digital front-end processing (gain / AGC, DC offset removal, synchronization, carrier / timing recovery) and finally symbol/bit demodulation using techniques specific to the modulation (e.g., BFSK, PSK, QAM).

These steps are illustrated in Figure 2.1. The diagram shows a general demodulation chain and signal examples for the BFSK modulation scheme.

Because of their specific use cases, the devices mentioned above are often designed with hardware architectures optimized for their applications. For instance, a smartphone contains dedicated radio chips for cellular communication, Wi-Fi, and Bluetooth (the latter two are frequently integrated into a single chip). These chips are tailored to handle particular modulation schemes and frequencies, delivering demodulated data to the device's processor.

Software-defined radio (SDR) is an approach where the indispensable RF and analog front end stays in hardware as close to the antenna as practical. This front

end covers antenna matching, low-noise and variable-gain amplifiers, quadrature mixers with local oscillator generation, analog filtering, automatic gain control loops, and the ADC and DAC clocking. After the signal is digitized the rest of the receive and transmit baseband chain moves into software or reconfigurable logic. That chain includes channel filtering, carrier recovery, timing recovery, demodulation, decoding and protocol handling [5].

In practice a radio front-end chip performs the analog tasks: filtering, gain control, frequency conversion, I/Q generation and conversion. It hands a complex digital stream to a programmable device (CPU, FPGA or both). That device carries out the flexible processing described in Subsection 2.1.3 and can be updated to support new modulation schemes and protocols without changing the RF hardware.

The developments and processing flow outlined in this section build upon material from the UCLouvain project courses LELEC2102 and LELEC2103 [6, 3].

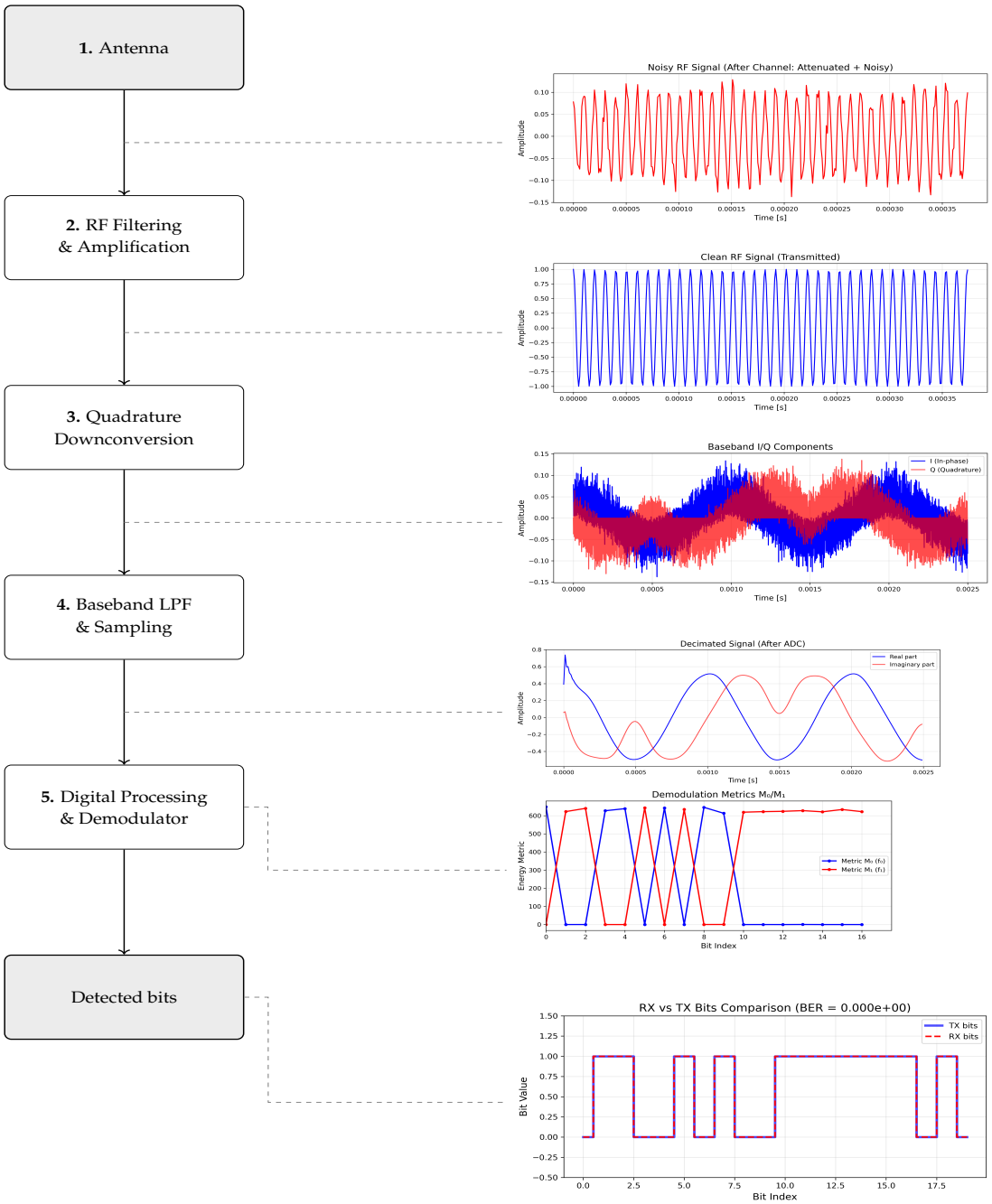

Figure 2.1 Generic demodulation chain with illustrative signal snapshots for a BFSK example.

2.1.2 Digital modulation schemes: BFSK / GMSK modulation

In the context of the Electronics Master's Project, the modulation scheme used is GMSK (Gaussian Minimum Shift Keying), which is a variant of MSK (Minimum Shift Keying) modulation.

First, the FSK (Frequency Shift Keying) modulation scheme is introduced. FSK is a digital modulation technique that uses discrete frequencies to represent different symbols. The real signal transmitted by the modulator can be expressed as:

$$s(t) = A\cos\big(2\pi f_c t + \phi(t)\big) \tag{2.1}$$

where:

- A is the amplitude of the signal,
- f_c is the carrier frequency (in Hz),
- $\phi(t)$ is the instantaneous phase (in radians).

In classic (non-continuous phase) FSK, the instantaneous phase is given by:

$$\phi(t) = 2\pi m\,\Delta f\,t, \quad nT \le t < nT + T. \tag{2.2}$$

where:

- m is the symbol value (e.g., $m \in \{0, 1\}$ for binary FSK),
- Δf is the frequency deviation (in Hz),
- T is the symbol duration (in seconds),
- n is the symbol index.

The instantaneous phase $\phi(t)$ jumps abruptly at symbol boundaries, causing spectral spreading.

Continuous-phase FSK (CPFSK), as used in GMSK, enforces smooth phase transitions to limit bandwidth expansion. In this case, the instantaneous phase is

given by:

$$\phi(t) = 2\pi + \int_{-\infty}^{t} d(\tau)d\tau, \tag{2.3}$$

$$\text{where} \quad d(t) = \Delta_f \sum_{n=-\infty}^{\infty} I(n)g(t - nT) \tag{2.4}$$

where:

- Δ_f is the frequency deviation (in Hz),
- $I(n)$ is the symbol at time nT (typically ± 1 for binary),
- $g(t)$ is the pulse shaping filter (rectangular for MSK, Gaussian for GMSK),
- T is the symbol duration.

The modulation index h is the frequency deviation normalized by the symbol rate. The modulation index h is defined as:

$$h = \frac{\Delta f}{f_b} \tag{2.5}$$

where f_b is the bit rate (in Hz) and Δf is the maximum frequency deviation from the carrier.
Focusing on the binary case (two symbols, $I(0)$ and $I(1)$), the modulation index becomes:

$$h = 2\Delta_f T \tag{2.6}$$

and induces a phase change of $\pm\pi h$ between each symbol period.

To minimize spectral spreading, the modulation index should be as small as possible. However, for good demodulation performance, the waveforms corresponding to each symbol should be orthogonal. The combination of these requirements leads to MSK (Minimum Shift Keying), a special case of CPFSK where the modulation index is exactly 0.5[1].

[1] In CPFSK $h = 2\Delta f T$. Orthogonality over one symbol occurs when $h = n/2$ with $n \in \mathbb{Z}$ (equivalently $2\Delta f = n/(2T)$). The smallest non-zero shift is $h = 0.5$ which gives MSK (minimum shift). Larger values like 1 or 1.5 remain orthogonal but widen the spectrum. Values not equal to $n/2$ lose orthogonality and introduce residual correlation.

Finally, GMSK is a special case of MSK where the filter $g(t)$ is Gaussian rather than rectangular. This introduces inter-symbol interference (ISI), which is absent in the previous cases, but improves spectral efficiency by reducing frequency sidelobes (Figure 2.2).

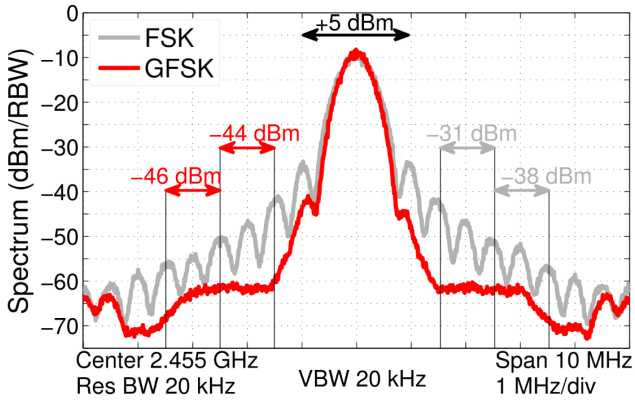

Figure 2.2 GMSK vs. MSK modulation waveform.

Source: [7]

Bandwidth for the project parameters. Here the bit and symbol rates are equal ($R_s = 50\,\text{ksym/s}$), so the symbol period is $T = 20\,\mu s$. For MSK ($h = 0.5$) the deviation satisfies $h = 2\Delta f T$, which gives

$$\Delta f = \frac{0.5}{2 \times 20\,\mu s} = 12.5\,\text{kHz}.$$

The two tones are therefore at $f_c \pm 12.5$ kHz (25 kHz spacing). A conservative upper bound on the occupied (double-sided) bandwidth is $B \approx 2(\Delta f + R_s) = 125\,\text{kHz}$ (Carson-style). MSK already concentrates more energy near f_c than discontinuous FSK. The Gaussian filter of GMSK narrows it further, with the exact narrowing set mainly by the filter's BT product[2]. Consequently a channel spacing slightly above 125 kHz is sufficient to carry the 50 kbit/s stream while leaving guard for filter roll-off.

[2]The Gaussian filter BT product is the 3 dB one-sided bandwidth B of the Gaussian filter multiplied by the symbol period T. Lower BT yields a tighter spectrum but increases inter-symbol interference.

2.1.3 Digital processing of I/Q signals

Once the I/Q components are retrieved, digitized, and sampled by the radio receiver chip (after the I/Q separation step described in Subsection 2.1.1), they can be processed to demodulate the signal. The demodulation process can be summarized in the following steps, where the first three steps are generic to most digital communication systems, while the final demodulation step is specific to the FSK modulation scheme used in this project:

1. Low-Pass Filtering (LPF):
The samples are filtered to remove high-frequency noise and retain the desired signal bandwidth. This step is common to all modulation types.

2. Transmission Detection:
The start of a transmission can be detected using an energy detection system. The receiver monitors the energy level of the received signal and detects when it exceeds a certain threshold, indicating the beginning of a transmission. This method is often used in conjunction with a preamble.

The preamble is a known sequence of bits transmitted before the actual data. It can be used without synchronization to detect the start of the transmission. Since the preamble is known by the receiver, it can be correlated with the received signal to identify the start of the transmission. This enables synchronization between the receiver and transmitter and allows retrieval of timing information.

3. Synchronization :
Carrier–frequency offset (CFO). Because the receiver clock is never exactly the same as the transmitter's, the center frequency suffers a small offset ε (in parts per million, ppm). After down-conversion, the IQ samples can be modeled as $r[k] = e^{j2\pi\varepsilon k}\, x[k] + w[k]$. In the context of the Electronics Master Project, the receiver estimates ε using Moose's algorithm, which relies on a preamble containing two identical parts of N symbols:

$$\hat{\varepsilon} = \frac{1}{2\pi N} \arg\Big(\sum_{k=0}^{N-1} r[k+N] r^*[k] \Big), \tag{2.7}$$

and multiplies the stream by $e^{-j2\pi\hat{\varepsilon}k}$ to recenter the tones.

Symbol-timing offset (STO). Even with the correct frequency, the sampling instants may cut each symbol at an unknown delay $d \in [0, L-1]$. Sampling too early or too late reduces the eye opening[3] and increases the bit error rate (BER). Oversampling enables selection of the optimal sampling instant. For each candidate delay d, the following metric is computed over the preamble:

$$E_d = \sum_n \left(|r_{1,d}(n)|^2 + |r_{2,d}(n)|^2 \right), \tag{2.8}$$

where $r_{1,d}$ and $r_{2,d}$ are the non-coherent correlations on the $+\Delta f$ and $-\Delta f$ branches, respectively. The optimum delay is $d^\star = \arg\max_d E_d$; from this point onward, all symbols are integrated at indices $k = nL + d^\star$.

Frame detection. After CFO and STO correction, the demodulator delivers a raw bit stream $b[n] \in \{0, 1\}$. To locate the exact start of the frame, the receiver searches for a known M-bit sync-word. The bits are first mapped to antipodal values $a[n] = 2b[n] - 1$, and a sliding correlator computes

$$C[m] = \sum_{i=0}^{M-1} a[m+i]s[i], \tag{2.9}$$

where $s[i] \in \{+1, -1\}$ is the bipolar version of the sync-word. When $|C[m]|$ reaches M (or exceeds a preset threshold), index m is taken as the first bit of the header. All following bits are then passed to the CRC[4] and payload decoder[5] with correct alignment.

4. Demodulation (project-specific: non-coherent binary FSK):

The following demodulation technique is specific to FSK modulation schemes and represents the approach used in this project.

[3] An eye diagram overlays many symbol-length segments of the received, post-filter waveform on a common one-symbol time axis. Its opening reflects the amplitude and timing margins for reliable decisions. With oversampling by L samples/symbol, an unknown fractional delay d shifts the decision instant away from the symbol center (where ISI is minimal). Sampling too early or too late closes the eye and increases the bit error rate.

[4] CRC (Cyclic Redundancy Check) is an error detecting code used to spot accidental changes in the received data.

[5] Payload: the useful transmitted data after the preamble, sync word and headers. The payload decoder extracts and interprets this content.

Let T be the symbol duration. Over each window $[0, T]$ the receiver forms the two inner products

$$r_1 = \frac{1}{2}\int_0^T r(t)\, s_1^*(t)\, dt, \qquad r_2 = \frac{1}{2}\int_0^T r(t)\, s_2^*(t)\, dt,$$

where $s_{1,2}(t)$ are the two orthogonal FSK waveforms. If s_1 was transmitted, $r_1 = \mathcal{E}e^{j\phi} + \nu_1$ and $r_2 = \nu_2$, ϕ being the unknown carrier phase and $\nu_i \sim \mathcal{CN}(0, \mathcal{E}N_0)$. Because a phase rotation does not change the magnitude, the MAP[6] decision reduces to a simple energy comparison:

$$\boxed{|r_1| > |r_2| \implies \hat{s} = s_1, \text{ else } \hat{s} = s_2}$$

—no carrier phase estimate is required.

2.2 Analysis of the current platform (LimeSDR Mini)

Before designing a new receiver platform, it is essential to thoroughly analyze the current solution, in this case the LimeSDR Mini. Understanding its technical specifications, hardware architecture, and operational strengths and limitations provides a solid foundation for developing a setup that matches or exceeds its capabilities. This analysis ensures that the new platform will deliver the required features, performance, and flexibility for educational use, while also addressing any shortcomings identified in the existing system.

2.2.1 General overview

The LimeSDR Mini platform consists of three main components: the LMS7002M transceiver, the Intel MAX 10 FPGA, and a USB 3.0 interface (Figure 2.3). The LMS7002M chip handles RF signal reception and transmission, converting analog signals to digital I/Q samples. These samples are processed and buffered by

[6] MAP (Maximum A Posteriori) is a statistical decision rule that selects the hypothesis with the highest posterior probability given the observed data. In digital communications, it is used to choose the most likely transmitted symbol based on the received signal.

the MAX 10 FPGA, which also manages configuration and control tasks.

Beyond buffering it implements a lightweight receive front end (FIR channel filtering plus an energy detector with a programmable threshold) so the FPGA only forwards I/Q bursts when a packet is detected instead of streaming continuously.

The USB 3.0 interface connects the board to a host computer, enabling high-speed transfer of I/Q data for further processing and visualization. This architecture provides a flexible and efficient environment for software-defined radio experiments, with the FPGA acting as a bridge between the RF front end and the host computer.

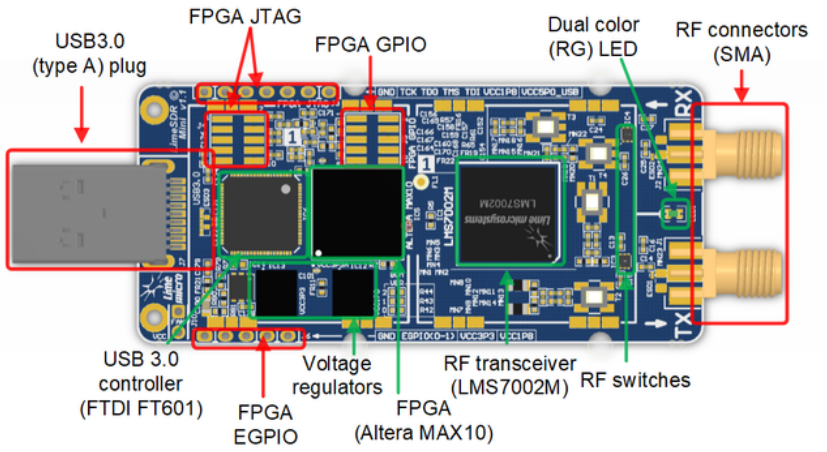

Figure 2.3 LimeSDR Mini overview.

Source: [8]

2.2.2 Technical specifications and hardware architecture

LMS7002M

The LMS7002M is a dual-transceiver (2 × TX + 2 × RX) that provides 12-bit resolution for both I and Q channels. Its integrated Transceiver Signal Processor (TSP) adds digital signal processing functions, such as digital gain adjustment, filtering, and sample rate conversion, to optimize the quality and format of the I/Q data delivered to the system. Table 2.1 lists the parameters relevant to this project.

Parameter	Specification
RF tuning range	0.1 MHz–3.8 GHz
Carrier frequency resolution	24.8 Hz @ 52 MHz reference
Instantaneous RF bandwidth	160 MHz (analog), 96 MHz (digital SISO), 48 MHz/ch (2×2)
ADC / DAC resolution	12 bit (ENOB ≈ 9 bit)
RX gain control range	89 dB (0.5–1 dB steps)
TX gain control range	70 dB (1 dB steps)

Table 2.1 Core LMS7002M specifications relevant to this work.

Source: [9]

Why these specifications matter:

- **RF tuning range:** The ability to tune from 0.1 MHz to 3.8 GHz ensures compatibility with a wide variety of radio standards. For this project, operation around 860 MHz is required, so the wide tuning range guarantees coverage of the target band.
- **Carrier frequency resolution:** Fine control (24.8 Hz steps) allows precise channel selection and frequency planning, which is essential for experiments involving multiple adjacent channels or custom waveforms.
- **Instantaneous RF bandwidth:** The available analog and digital bandwidths determine how much of the spectrum can be acquired at once. Sufficient bandwidth is needed to capture the entire useful signal without excessive out-of-band noise or interference.
- **RX gain control range:** The 89 dB gain span is the sum of all adjustable gain stages distributed along the receiver signal chain. Each stage has a distinct function. Some amplify weak signals at the input, others adjust levels before digitization, and some provide digital scaling after conversion. This combined adjustment enables the receiver to handle a wide range of signal types and power levels, optimizing performance for different scenarios.

AGC. An on-chip automatic-gain-control (AGC) engine can adjust the RX gain in real time, but remains unused in the current setup.

Package. The device comes in a compact 11 mm × 11 mm, 324-ball BGA package, well-suited for dense RF layouts.

I/Q Interface. Baseband I/Q can be accessed either as differential analog outputs or via a 12-bit parallel LimeLight™ digital I/Q interface that uses single-ended LVCMOS signaling for direct connection to an FPGA or SoC. In the LimeSDR Mini, the parallel digital output is used to interface with the FPGA [9].

The simplified receiver signal chain of the LMS7002M is illustrated in Figure 2.4. The incoming RF signal is first amplified by a low-noise amplifier (LNA), then down-converted to baseband by a quadrature mixer. The resulting I/Q signals are further amplified and filtered to remove unwanted frequency components. A programmable gain amplifier (PGA) adjusts the signal level before it is digitized by a pair of analog-to-digital converters (ADCs). Finally, a digital signal processor (TSP) performs additional processing and formatting before sending the data to the FPGA.

This architecture enables flexible gain control, effective filtering, and direct digital access to the received baseband signal. For clarity, internal feedback paths such as AGC and loopback are not shown in the figure.

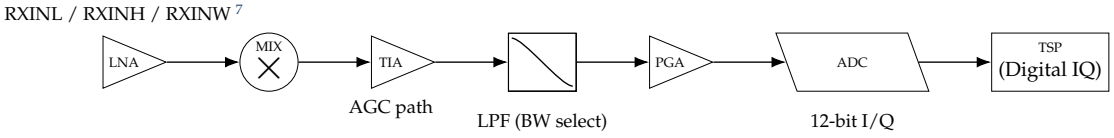

Figure 2.4 Simplified receive path in the LMS7002M (single channel).

Altera MAX 10 FPGA

The LimeSDR-Mini embeds an Intel *MAX 10* device (part number 10M16SAU169C8G). It supplies just enough logic, memory, and DSP to move and pre-process the incoming I/Q stream while acting as the USB-3.0 bridge.

[7] RXINL, RXINH, and RXINW are the three alternative RF input ports of the LMS7002M; select the appropriate one based on the intended carrier band (low vs. high/wide) and the external matching network [9].

Resource	Quantity
Logic elements (LE)	15 840
Embedded SRAM (M9K)	549 kbit
On-chip flash (user + config)	2.368 Mbit
DSP blocks (18 × 18)	45
Fractional-N PLLs	1
Package	169-ball UBGA, 11 mm × 11 mm

Table 2.2 Intel MAX 10 resources available on the LimeSDR-Mini.

Source: [10]

Why these resources matter (receive path only):

- **Logic elements** hold the state machines, interface the LMS7002M baseband I/Q streams, manage the USB bridge (FX3/GPIF), and host the embedded *Nios II* soft processor.
- **DSP blocks** implement a 31-tap complex FIR filter and a real-time energy estimator; using the hard 18 × 18 multipliers saves thousands of LEs.
- **M9K memories**[8] provide the storage required for samples and intermediate processing results, as well as FIR coefficients and detection thresholds.
- A minimalist **NIOS II soft CPU** (about 2 k LE) configures the LMS7002M and exposes run-time registers over USB.

LMS7002M–FPGA interface (RX only)

The LMS7002M exposes a 12-bit parallel LimeLight™ digital I/Q interface using single-ended 2.5 V LVCMOS. In receive mode, the transceiver drives `MCLK`, `IQSEL`, and `DIQ[11:0]` toward the MAX 10.

I and Q samples are multiplexed on `DIQ[11:0]` and identified by `IQSEL`. The FPGA captures the bus in DDR or SDR, demultiplexes it into 12-bit I and 12-bit Q words, time-stamps the samples, and writes them into a dual-clock FIFO to cross into the internal processing domain [9].

[8] The M9K memories are 9 Kbit blocks of RAM.

Configuration path. At power-up a tiny Nios II soft-core inside the FPGA boots from the on-chip flash and loads a default register set into the LMS7002M through the dedicated four-wire SPI bus (`FPGA_SPI_SCLK`, `MOSI`, `MISO`, `LMS_SS`). During operation, the PC can send control packets over USB; the FPGA firmware forwards them to the Nios II, which transparently rewrites the target LMS registers via the same SPI link, making every parameter hot-configurable without disrupting the received data stream.

FPGA to PC connection

The host computer runs GNU Radio powered by a GNU/Linux-based OS. The whole configuration and control of the LimeSDR-Mini is achieved through specific modules in GNU Radio. The I/Q samples are sent to the host computer over the USB interface. The demodulation process and the display of the results are done in GNU Radio, which provides a user-friendly graphical interface to visualize the data flow and the demodulated signals. In the legacy laboratory setup that this project aims to replace, the LimeSDR Mini was typically operated at a complex baseband sampling rate of 400 kS/s.

Strengths and limitations

The LimeSDR-Mini provides a genuinely *plug-and-play* workspace for radio experimentation. A single USB 3.0 link powers the board and carries the I/Q stream, so bench setups stay neat and can be reproduced in minutes. Its broadband transceiver spans most commercial and license-free bands, and the internal signal path accepts wide instantaneous bandwidths that comfortably exceed the needs of typical laboratory exercises. Because the platform is supported by open-source tools such as `LimeSuite` and `GNU Radio`, students can move from waveform synthesis to visualization without proprietary barriers or additional licenses [11, 9]. The on-board FPGA can be configured through `Quartus`, which is well known by the students from their previous courses.

These advantages come with notable drawbacks.
First, the on-board MAX 10 FPGA offers only limited logic resources. This limitation restricts hardware acceleration and forces most time-critical processing to be performed on the host computer [10]. As a result, students have less opportunity to experiment with real-time digital signal processing directly on the FPGA, which can reduce the pedagogical value of hardware-based learning.

Second, the highly integrated build with fine-pitch BGAs on a dense multilayer PCB makes component-level repair unrealistic in an academic workshop. Third, production of the original LimeSDR-Mini has been suspended. Replacement units and spare parts are therefore increasingly difficult to obtain [4]. Finally, because much of the RF and digital architecture is fixed, learners have only partial insight into the board's internal operation and little room to adapt it for alternative designs.

In summary, the LimeSDR-Mini delivers a compact, wideband SDR front-end that is ideal for rapid demonstrations. However, its limited FPGA capacity, non-serviceable hardware, and uncertain availability highlight the need for a more flexible and sustainable solution, as discussed in the following chapter.

2.3 Technological choices for the new platform

2.3.1 FPGA platform: DE0-Nano vs. DE10-Nano

One of the requirements of this work was to reuse available hardware resources from UCLouvain while keeping the use of `Quartus` as the FPGA development tool. This left us with two options: the DE0-Nano and the DE10-Nano boards.

DE0-Nano. The DE0-Nano is a small, low-cost board with a Cyclone IV FPGA. It is already used in other courses in the EPL curriculum, which makes it a familiar choice for students. The version of the Cyclone IV embedded in the DE0-Nano shows comparable resources to the MAX 10 FPGA, and even surpasses it in some aspects, such as the number of logic elements and the number of DSP blocks. It could be a good candidate for this work.

DE10-Nano. On the other hand, the DE10-Nano is a more powerful board with a Cyclone V FPGA. It has more logic elements, more DSP blocks and more RAM than the DE0-Nano. More interesting, it has a dual-core ARM Cortex-A9 processor that can run Linux, which opens up new possibilities for software-defined radio applications. It also embeds more interfaces, such as HDMI and Ethernet.

Both devices are compared in Table 2.3.

	DE10-Nano	DE0-Nano
FPGA device	Intel Cyclone V SE 5CSEBA6	Intel Cyclone IV EP4CE22F17C6N
Logic elements	110,000	22,320
DSP blocks (18 × 18)	112	66
Embedded RAM (FPGA)	5.6 Mbit	0.6 Mbit
On-board external memory	1 GB DDR3 (shared with HPS)	32 MB SDRAM
Processor core	Dual-core ARM Cortex-A9 HPS	— (soft Nios II optional)
Ethernet	1× Gigabit Ethernet PHY	—
HDMI	1× HDMI transmitter	—

Table 2.3 High-level comparison of DE10-Nano and DE0-Nano development boards.

Source: [12, 13, 14]

Comparison

Since the radiotransceiver chip might differ from the one used in the LimeSDR-Mini, the implemented logic might be slightly different in certain points in terms of data rate, configuration of the radio chip and communication with the host computer. Since the original setup almost fulfills the MAX 10 FPGA capacity in terms of logic elements and uses all the PLL's, a new design slightly different from the original might overload the DE0-Nano FPGA. (Table 2.4)

Another point that needs to be considered is the communication between the FPGA and the host computer. The LimeSDR mini included a USB 3.0 interface, which is not available on the DE0-Nano board. An additional parallel CMOS/USB adapter would be needed to connect the DE0-Nano to the host computer, which would add complexity to the design and increase the cost of the solution.
In contrast, the DE10-Nano board includes a Gigabit Ethernet interface. This allows for high-speed data transfer and simplifies the connection to the host computer, making it a more attractive option for software-defined radio applications.

The only drawback of the DE10-Nano is that it needs to be powered through a barrel jack, which is not as convenient as the mini-USB connector of the DE0-Nano. However, this is a minor inconvenience compared to the advantages of the DE10-Nano.

Resource	Used	Available	Utilization
Logic elements	13,626	15,840	86 %
9-bit multipliers	3	90	3 %
DSP blocks[9]	2	45	4 %
Memory bits	244,624	562,176	44 %
PLLs	1	1	100 %

Table 2.4 Resource utilization of the MAX 10 device 10M16SAU169C8G.

Source: Original LimeSDR-Mini design, Quartus Prime 18.1 report

Conclusion

In conclusion, as the price needs to be considered, the DE10-Nano seems to be the best choice, avoiding the need for an additional USB interface and providing more resources for the design.

2.3.2 Software tools: signal processing and visualization

GNU Radio is the software currently used with the LimeSDR-Mini setup for every stage of the receive chain. The open-source toolkit provides a graphical interface that lets students design, implement and visualize synchronization, demodulation, and other baseband tasks. Its large user community and active development make it a natural choice for teaching.

The educational team has added custom GNU Radio blocks that automate LimeSDR-Mini configuration and supply the demodulators required in the laboratory sessions.

[9] Each MAX 10 embedded multiplier block can implement *either* one 18x18 *or* two 9x9 multipliers [15]. Hence, the three 9x9 multipliers reported by Quartus occupy two DSP blocks out of the 45 available.

Other environments are available. MATLAB Simulink and LabVIEW offer extensive DSP libraries but require paid licenses, increasing the cost for an academic program. A pure-Python stack based on NumPy and SciPy is license-free, yet it lacks an integrated streaming framework and visual editor comparable to GNU Radio. LabVIEW can operate with non-National Instruments hardware, including future boards, but doing so involves additional drivers and wrapper layers that complicate integration.

Because none of these alternatives provides clear advantages over GNU Radio, and in view of the bespoke blocks already in place, GNU Radio will remain our tool of choice for signal processing and visualization on the new platform.

2.3.3 Radio transceiver chip: selection criteria and comparison

Requirements

The transceiver chip is a key element of the future software-defined radio platform; it must satisfy several constraints dictated by the laboratory syllabus.

1. Operation in the target band centered on 860 MHz[10].

2. Delivery of *raw* I/Q samples[11] so that all demodulation is performed in GNU Radio.

3. A gain-control mechanism, either automatic (AGC) or manual.

4. An I/Q interface compatible with the FPGA, implemented either as a parallel bus or as a high-speed serial link.

5. Digital configuration from the FPGA (frequency, gain, sample rate, filter bandwidth, etc.).

[10]The exact center frequency may vary slightly from one student group to the next, depending on the channel allocation.

[11]"Raw" means un-demodulated complex baseband data; the signal is nevertheless filtered, down-converted and, if necessary, decimated on-chip to reduce the data rate and improve the SNR.

6. Although the present work focuses on reception, the ability to transmit is desirable for future projects.

7. Programmable channel bandwidth in order to match the chosen modulation scheme with a suitable SNR.

Candidates

These requirements naturally point to transceiver chips intended for software-defined radio: such devices are among the few capable of outputting raw I/Q data while exposing flexible gain control.

In contrast, most commercial radio front-ends target a single application and embed full demodulation and baseband processing, which makes them unsuitable here.

Table 2.5 summarizes the main candidates identified during the survey phase, selected for their availability, electrical specifications, and compatibility with the DE10-Nano board.

Transceiver	Freq. range	BW (MHz)	I/Q interface	Pkg.	Typical apps
AD9361	0.07–6 GHz	0.2 - 56	CMOS/LVDS	BGA	SDR, IoT, wireless
LMS7002M	0.1 MHz–3.8 GHz	0.7 - 56	CMOS/LVDS	aQFN	SDR, MIMO
AD9363	0.325–3.8 GHz	0.2 - 20	CMOS/LVDS	BGA	SDR, wireless
AT86RF215	389–510, 779–1020 MHz; 2.36–2.48 GHz	0.16 - 4	LVDS serial	QFN	Low-cost IoT

Table 2.5 Key characteristics of the transceiver candidates.

Source: [16, 17, 9, 18]

The three first candidates are all designed for SDR applications, while the last one is a low-cost transceiver for IoT applications, but which has the particularity of being able to output raw I/Q samples, making it a suitable candidate for low-cost SDR applications.

A noticeable difference between the different candidates is the package type. The AD9361 and AD9363 from Analog Devices are available in BGA packages, which

are more difficult to solder and require a PCB with a fine pitch. The LMS7002M is available in an aQFN package, which is easier to solder and requires less space on the PCB, but is still quite complex to solder. The AT86RF215 is available in a QFN package, which is the easiest one to solder as all the pins are accessible from the sides.
A visual comparison of these package types is shown in Figure 2.5.

(a) BGA **(b)** aQFN **(c)** QFN

Figure 2.5 Comparison of the main package types for candidate transceivers.

Final choice

The Microchip AT86RF215 meets every technical requirement of the receiver: it covers the 860 MHz band, delivers raw LVDS I/Q data, offers programmable gain, and supports channel bandwidths from 160 kHz up to 4 MHz—well above the required bandwidth for this project (125 kHz) [12].

Its unit price is roughly ten times lower than broadband SDR parts such as the AD9361, AD9363 or LMS7002M because it targets high-volume IoT markets, embeds slower ADCs and narrower filters, and integrates a single RF chain; the die is therefore smaller and the production yield higher. The 48-pin QFN package can be soldered on a standard four-layer board without microvias or BGA reflow, further reducing cost and assembly risk.

In short, the extra bandwidth, MIMO capability and 6 GHz coverage of the larger SDR transceivers are unnecessary for our modulation scheme, whereas the AT86RF215 delivers all required features at a fraction of the price and complexity.

[12]The on chip filter only represents a first stage of filtering for reducing the noise and the bandwidth of the signal. The final filtering setting the real bandwidth is done in GNU Radio.

The simplified receiver signal chain of the AT86RF215 is shown in Figure 2.6. The incoming RF signal is first amplified by a low-noise amplifier (LNA) and mixed down to a *low intermediate frequency* (low-IF). A selectable analog channel filter at that low IF rejects adjacent-channel and out-of-band energy before a sigma-delta ADC samples the signal (internal sampling at 32 MHz per the datasheet).

The digitized low-IF stream then enters the receive digital front end (RX_DFE), which decimates it to 4 MS/s and performs a digital quadrature down-conversion to true zero-IF (baseband). Additional decimation and filtering stages further narrow the bandwidth and shape the passband. The resulting complex I/Q samples are finally serialized and sent to the FPGA over the LVDS link. For clarity, internal feedback paths and AGC loops are not shown.

Compared to the LMS7002M, the AT86RF215 features a much simpler analog chain, with only a single amplification stage (LNA) before digitization. This design reduces power consumption and complexity, but offers a more limited dynamic range and less flexibility for handling strong interferers. In practice, this is well suited to narrowband IoT applications, where received signal levels are more predictable.

In contrast, the LMS7002M includes three analog gain stages, providing a much wider dynamic range and finer control, which is essential for broadband and MIMO systems but comes at the cost of higher complexity and power consumption.

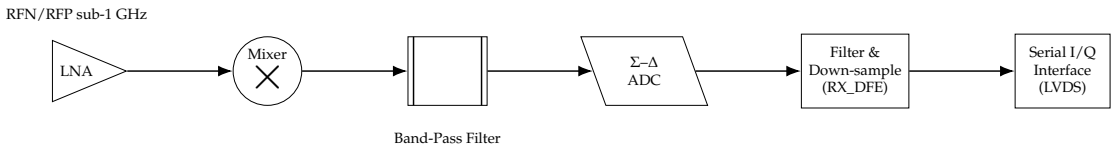

Figure 2.6 Simplified sub-1 GHz receive path in the **AT86RF215**.

2.3.4 RX Shield: a custom PCB?

What a custom shield would require

To interface the AT86RF215 with the DE10-Nano we initially envisaged a dedi-

cated *RX shield*[13]. Such a board would have to provide every support function that the bare transceiver needs:

- *Power generation* — low-noise converters for 3.3 V, 1.8 V and 1.2 V, plus ample decoupling near the chip.
- *Reference clock* — a 26 MHz crystal (or TCXO) with its load network; an optional 32 kHz crystal for the sleep timer.
- *RF front-end* — a sub-GHz balun[14] , matching network and a 50 Ω microstrip trace ending in an SMA connector, with ground-via stitching for shielding.
- *LVDS data path* — one differential pair carrying the serialized complex I/Q stream plus one differential clock pair, PCB traces length and impedance matched.
- *Control interface* — an SPI bus, reset lines, GPIOs and level shifters pinned to a mezzanine or GPIO header compatible with the DE10-Nano.
- *Loop filter and PLL passives* — components sized for the sub-GHz VCO range, placed close to the transceiver.
- *Debug facilities* — test points for supply rails and SPI, an SMA footprint for an external reference clock and space for optional EMI shielding can.

[13] A "shield" is an add-on circuit board that plugs into a main development board (such as an FPGA or microcontroller), providing additional functions or interfaces—commonly used in prototyping and education.

[14] A balun (balanced–to–unbalanced transformer) converts between a differential (balanced) antenna or differential RF path and a single-ended (unbalanced) circuit. It may also provide impedance transformation and improves common-mode rejection.

Designing, laying out and fabricating such a controlled-impedance PCB, followed by one or two prototype spins, would add several weeks and a non-trivial budget to the project.

A ready-made alternative
During the component survey we discovered the *ATREB215-XPRO-A* evaluation module (Figure 2.7) [19, 20]. This board already integrates the AT86RF215, on-board regulators, the 26 MHz crystal, a fully tuned sub-GHz RF path with SMA connector and a 20-pin header exposing LVDS I/Q, SPI and power lines. Adopting the XPRO-A therefore removes the RF layout risk, shortens hardware bring-up to a few hours and lets the course concentrate on baseband signal processing rather than PCB debugging.

Figure 2.7 The ATREB215-XPRO-A evaluation board, integrating the AT86RF215 transceiver and all necessary support circuitry.

Source: [19]

2.3.5 Final system architecture

Based on the technological choices outlined above, the final platform consists of the following key components:

RF front-end: The ATREB215-XPRO-A evaluation board, which integrates the AT86RF215 transceiver along with all necessary RF circuitry, power regulation, clock generation, and signal conditioning. This ready-made solution eliminates the need for custom PCB development.

FPGA platform: The DE10-Nano development board, featuring a Cyclone V FPGA and dual-core ARM Cortex-A9 HPS. This board provides the digital processing capabilities and connectivity options required for the platform.

Interface connections: Power and SPI travel through the XPRO 20-pin header on the XPRO-A into the DE10-Nano, while the LVDS differential clock and data pair are taken from the soldered pads/pins on the XPRO-A and routed to the matching GPIO header pins on the DE10-Nano.

host computer: A standard computer running GNU Radio, connected to the DE10-Nano via standard Gigabit Ethernet cable for data exchange and system control.

Software tools: GNU Radio serves as the signal processing and visualization environment, maintaining compatibility with existing educational materials and laboratory sessions.

3

Platform Design and Development

3.1 Overall system architecture

3.1.1 General overview

In the Section 2.3, we discussed the choice of the different components that would be used in the new platform. This chapter explains how these components are integrated into a complete system, from the antenna to the host computer. The proposed platform integrates three main subsystems, illustrated in Figure 3.1.

At the RF front-end, the system uses the **REB215-XPRO-A** evaluation board from Atmel, which embeds the dual-band **AT86RF215** transceiver. Only the sub-1 GHz interface is used, connected via a dedicated SMA port. The chip handles RF amplification, downconversion and I/Q digitization, then streams 32-bit LVDS I/Q data to the FPGA. All configuration registers are accessed via a standard SPI interface through the GPIOs of the DE10-Nano board.

The HPS runs a **Debian GNU/Linux** system (based on the public `zangman/de10-nano` Git repository [21]), which handles the high-level logic and SPI communication with the radio transceiver for configuration. Processed baseband samples are streamed in real time to the host computer over the integrated **Gigabit Ethernet** link.

The host system is a general-purpose computer running **GNU Radio 3.10** under Linux. It receives the filtered and digitized baseband signal, demodulates it in software, and optionally displays the decoded packets or stores them for further analysis.

The configuration of the whole system (from RF front-end to host computer) is specified in the GNU Radio interface. The configuration is synchronized with the HPS every time the GNU Radio flowchart is executed, which in turn configures the radio transceiver.

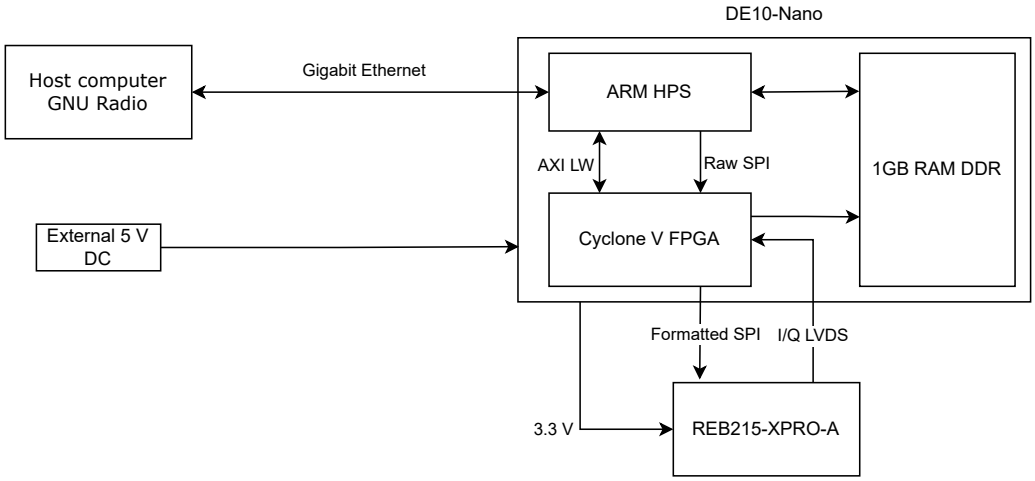

Figure 3.1 Overall system architecture.

3.2 Transceiver integration and configuration

3.2.1 SPI configuration interface of the AT86RF215

The transceiver exposes a *byte-oriented* SPI slave with four signals `SCLK`, `MOSI`, `MISO`, and `SELN` (active low) [18]. It uses **SPI Mode 0** (CPOL = 0, CPHA = 0)[1]:

[1] *Clock polarity* (CPOL) indicates the idle level of `SCLK` — 0 means logic-low. *Clock phase* (CPHA) defines when data are sampled; CPHA = 0 means the slave samples `MOSI` on the first (rising) edge and shifts `MISO` out on the subsequent falling edge.

MOSI is sampled on the rising edge of SCLK, whereas MISO changes on the falling edge[18]. The slave tri-states MISO while SELN is high.

Command structure and access cycles. Figure 3.2 illustrates the three-byte *single write* cycle, while Figure 3.3 shows the corresponding *single read* cycle. The first two bytes combine the 2-bit MODE field and the 14-bit register address; the third byte is either the data to be written (**write**) or the value returned on MISO (**read**).

- MODE = 00 — register **read**
- MODE = 10 — register **write**
- 01 and 11 are reserved [18].

SELN must remain low for the entire three-byte sequence; keeping it low after the third byte initiates a *block access* in which additional data bytes map to consecutive addresses (ADDRESS + 1, + 2, . . .) [18].

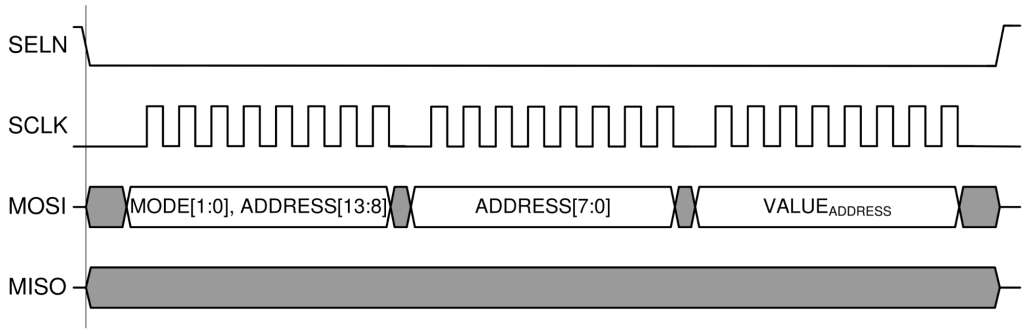

Figure 3.2 SPI single *write* access; SELN remains low for the entire three-byte frame.

Source: AT86RF215 datasheet Figure 4-2, [18]

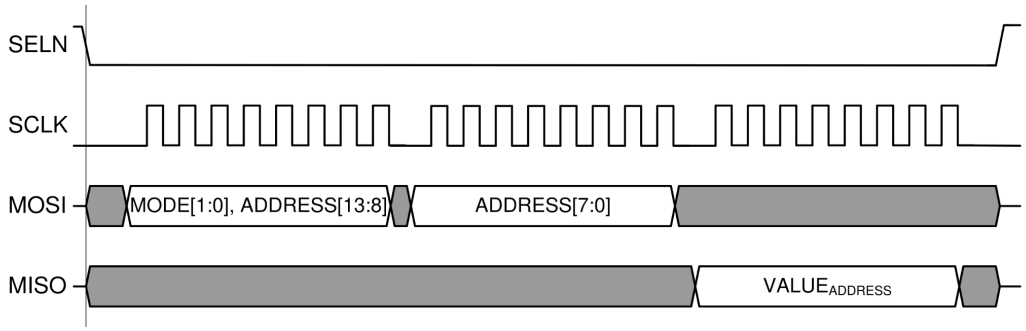

Figure 3.3 SPI single *read* access; `SELN` remains low for the entire three-byte frame.

Source: AT86RF215 datasheet Figure 4-3, [18]

Electrical and timing notes.

- **Clock rate** — any `SCLK` up to the datasheet limit is allowed; the command is latched on the *last* rising edge [18].
- **Setup/hold** — `MOSI` must be valid at least 5 ns before/after the rising edge; `MISO` is valid 5 ns after each falling edge [18].
- **Inter-frame gap** — once `SELN` returns high, the bus must remain idle for at least 50 ns before the next access starts [18].

3.2.2 LVDS DDR data transmission

Introduction to LVDS data transmission

Low-Voltage Differential Signalling (LVDS) is an electrical interface standard defined in ANSI/TIA/EIA-644-A [22]. Two complementary conductors carry a constant current source that produces a typical ±350 mV swing across a 100Ω termination placed at the receiver input. The low amplitude, combined with a common-mode bias of about 1.2 V, minimizes electromagnetic radiation, allows data rates well beyond 400 Mbit/s on standard FR-4 traces, and keeps dynamic power dissipation below 10 mW at tens of MHz [23].

LVDS itself specifies only the physical layer: voltage levels, termination, and fail-safe behavior. Above that, designers are free to choose a serialization scheme

(single-data-rate (SDR), double-data-rate (DDR), 8b/10b-encoded links, etc.) and to decide whether the clock is embedded (e.g. with CDR[2]) or forwarded on a separate differential pair.

LVDS DDR data transmission in the AT86RF215

Clock and data rate. The transceiver drives one differential clock pair (`RXCLKP/N`) at 64 MHz and one differential data pair (`RXD09_P/N`). With double-data-rate (DDR) transfer, one 32-bit word is launched on every 16 clock edges, yielding a fixed link rate of 4 MS/s [18]. When the user selects a sampling frequency below 4 MS/s via the `RXDFE.SR` register, the device inserts idle *zero-words* so that the LVDS timing remains unchanged [18].

Word format. Each 32-bit frame contains both I and Q information:

- bits 31:30 – synchronization marker `10` for the I field;
- bits 29:16 – signed 13-bit I sample (bit 16 is always 0);
- bits 15:14 – synchronization marker `01` for the Q field;
- bits 13:0 – signed 13-bit Q sample.

This "10/01" marker pair lets the FPGA detect word boundaries without needing an external framing signal [18].

Timing adjustment (`IQIFC1.SKEWDRV`). Data are center-aligned to the forwarded clock; the datasheet specifies an asymmetrical setup/hold window of -1 ns of setup time and $+2$ ns of hold time with respect to each clock edge (Figure 3.4) . Register `IQIFC1.SKEWDRV` offers four discrete delay settings (1.906 to 4.906 ns in steps of 1 ns) to compensate PCB or package skew between the clock and data pairs [18].

Electrical characteristics. The AT86RF215 supports two distinct LVDS electrical modes that share the same protocol (DDR 64 MHz, 32 bits/word) but differ in their electrical characteristics:

[2]Clock and Data Recovery (CDR) circuits extract the sampling clock from data transitions (often using a PLL or DLL). This removes the need for a separate forwarded clock pair and line coding (e.g. 8b/10b) maintains transition density.

- **Proprietary SLVDS mode:** Low-swing, low common-mode voltage mode designed for reduced power consumption. The common-mode voltage is configurable (150–300 mV) via the `IQIFC0.CMV` register (see Appendix A.2), with adjustable drive current (1–4 mA).
- **IEEE 1596.3 LVDS mode:** Standard LVDS mode for interoperability with conventional LVDS receivers. The common-mode voltage is fixed at 1.2 V and is activated by setting `IQIFC0.CMV1V2 = 1`.

In this project, the transceiver is configured to operate in **IEEE 1596.3 mode** with a common-mode voltage of 1.2 V to ensure compatibility with the FPGA's standard LVDS receivers. The link is DC-coupled and must be terminated with a single 100 Ω differential resistor at the FPGA receiver [18].

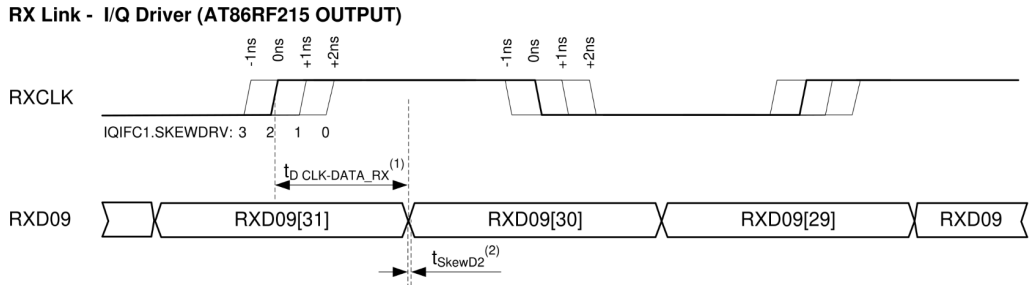

Figure 3.4 Timing of the receive LVDS link in DDR mode.

Source: AT86RF215 datasheet [18]

3.2.3 Configurable parameters and operational modes

The AT86RF215 supports multiple operational modes and exposes a wide range of configuration registers to control its internal behavior. This includes the selection of RF bands, signal path (RX or TX), operating frequency, sampling rate, analog filtering bandwidth, and I/Q interface properties. Although the chip features dual-band capabilities and integrated PHY-level packet demodulation, this project restricts its use to the **sub-GHz (RF09) path** in **raw I/Q streaming mode** to meet the project requirements.

Available transceiver modes

- **RF front-ends:** The transceiver embeds two independent radio paths:

  - RF09 — covers 389–510 MHz and 779–1020 MHz
  - RF24 — covers 2360–2483.5 MHz

  Only the RF09 path is used in this platform.

- **I/Q mode vs. demodulated mode:** The chip can operate either in I/Q streaming mode (CS.MODE = 0b00) or in baseband demodulation mode. In this project, only I/Q streaming mode is used; the internal demodulation function of the chip is disabled.
- **Operating states:** Each RF chain can be set independently to one of several states by writing a command to the RFn_CMD.CMD[2:0] register. The current state can be read from RFn_STATE.STATE[2:0] [18]. The complete finite state machine diagram is provided in Appendix A.1, and the main states are:
  - **TRXOFF:** Transceiver off (RF inactive, SPI accessible). Default after power-on or wake-up; reachable from any state.
  - **TXPREP:** Preparation for TX/RX; analog power and PLL enabled, calibrations run. Signals readiness with IRQ.
  - **TX:** Transmission active; PLL set to TX frequency, TX front-end and digital chain enabled.
  - **RX:** Reception active; PLL set to RX frequency, RX front-end and digital chain enabled (I/Q interface active in I/Q mode).
  - **TRANSITION:** Temporary state during mode changes.
  - **RESET/SLEEP:** Transceiver in reset or sleep; returns to TRXOFF after exit. SLEEP resets registers but keeps digital regulator on.
  - **DEEP SLEEP:** Internal deep-sleep mode (not a register code); minimal power, entered when both RF chains are in SLEEP.
  - **POWER OFF:** Internal state when chip is powered down; next power-on leads to TRXOFF.

Registers used in the project

Overview. The control software configures the transceiver during setup by writing to specific configuration registers. These registers control the RF channel selection, analog and digital filter bandwidths, I/Q output mode, AGC behavior, and LVDS interface characteristics. The complete list of configuration registers used in this project is provided in Appendix A.2.

3.2.4 Pinout and Connections

SPI and Power Supply

The REB215-XPRO-A board features a 2×10-pin XPRO expansion connector, shown in Figure 3.5. Its key characteristics are summarized in Table 3.1. The connector carries:

- **Power rails** 3.3 V and GND
- **SPI bus** `SCLK`, `MOSI`, `MISO`, `SELN` (active low)
- **Interrupt** `IRQ`
- **Reset** `RSTN` (active low)
- **GPIO** user-defined general-purpose pins

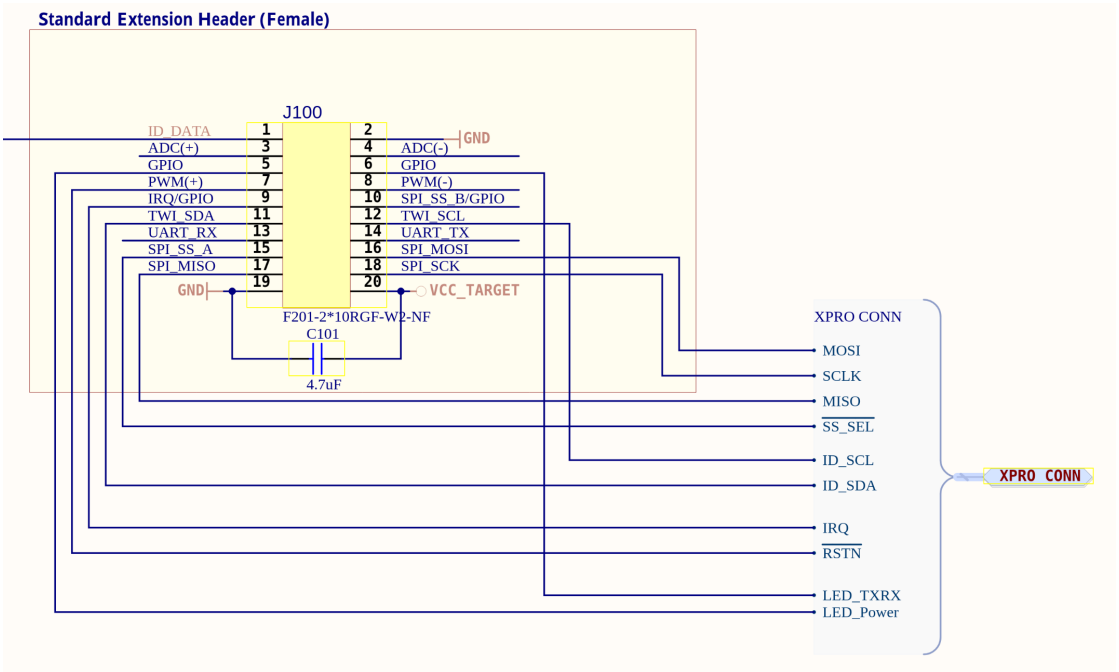

Figure 3.5 XPRO-A connector: schematic view and pin mapping.

Source: REB215XPROA design documentation [24]

Parameter	Value
Number of positions	10
Pitch	2.54 mm (0.1 in)
Number of rows	2
Row spacing	2.54 mm (0.1 in)
Contact gender	Female

Table 3.1 Mechanical characteristics of the XPRO connector (REB215-XPRO-A board).

LVDS Interface

The LVDS data and clock pairs are routed to an unpopulated, SATA-style edge footprint on the PCB (Figure 3.6). Four wires have been soldered to this footprint to access the signals externally:

- **Data pairs** : `RXD09_N`, `RXD09_P`
- **Clock pairs** : `RXCLK_N`, `RXCLK_P`

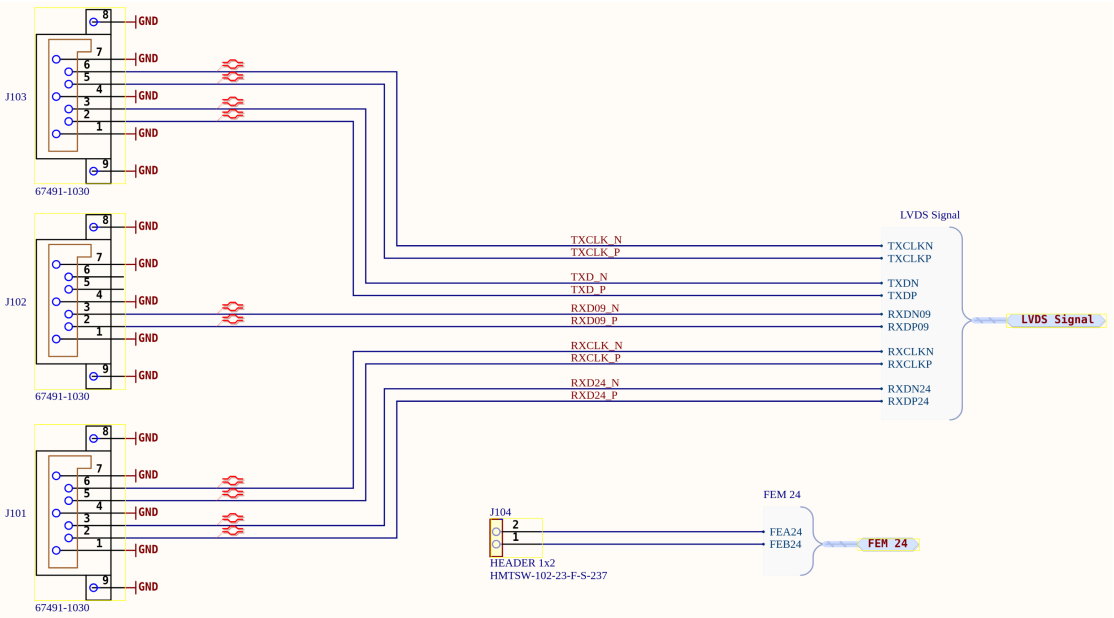

Figure 3.6 LVDS interface—SATA-style footprint on the REB215-XPRO-A PCB.

3.3 FPGA/HPS Embedded HDL/Software Development

3.3.1 FPGA Architecture and PS/PL Partitioning

The **DE10-Nano** provides several key functions for this work, distributed between the Hard Processor System (HPS) and the Programmable Logic (PL) according to the nature of each task and available resources. As introduced in Section 3.1, the HPS runs a full Debian GNU/Linux system, enabling the use of standard GNU tools, daemons, and drivers—greatly simplifying software development and integration. Built-in functions can be leveraged directly in software, avoiding the need to reimplement them in HDL.

The Quartus project was initialized from the Golden Hardware Reference Design (GHRD) produced by the Intel System Builder (template: `DE10_NANO_SoC_GHRD`), which supplies the baseline HPS–FPGA clocks, resets, bridges and peripheral scaffolding on which the custom logic was added.

Conversely, the PL is used to handle real-time operations and tasks that are either impractical or too resource-intensive for the HPS. The instantiated HDL IPs and their connections to the HPS are shown in the FPGA block diagram (see Figure 3.8[3]).

Hard Processor System (HPS). Running a full Debian GNU/Linux distribution, the HPS provides the standard environment required for user-space daemons, drivers, and network services. A lightweight systemd-managed daemon listens on a UDP socket for an incoming configuration file and (re)launches the C streaming program with updated parameters. A dedicated C program

- parses the configuration received from the host computer,
- programs the AT86RF215 transceiver through the SPI driver,
- orchestrates the DMA[4] transfers between the PL and the DDR3,

[3] On this Figure and for the rest of the manuscript, the clock and reset signals are not always explicitly represented.

[4] Direct Memory Access controller, able to write to / read from DDR3 without CPU intervention.

- forwards the I/Q samples to the host via UDP over the on-board Gigabit Ethernet.

Figure 3.7 illustrates the software flowchart.

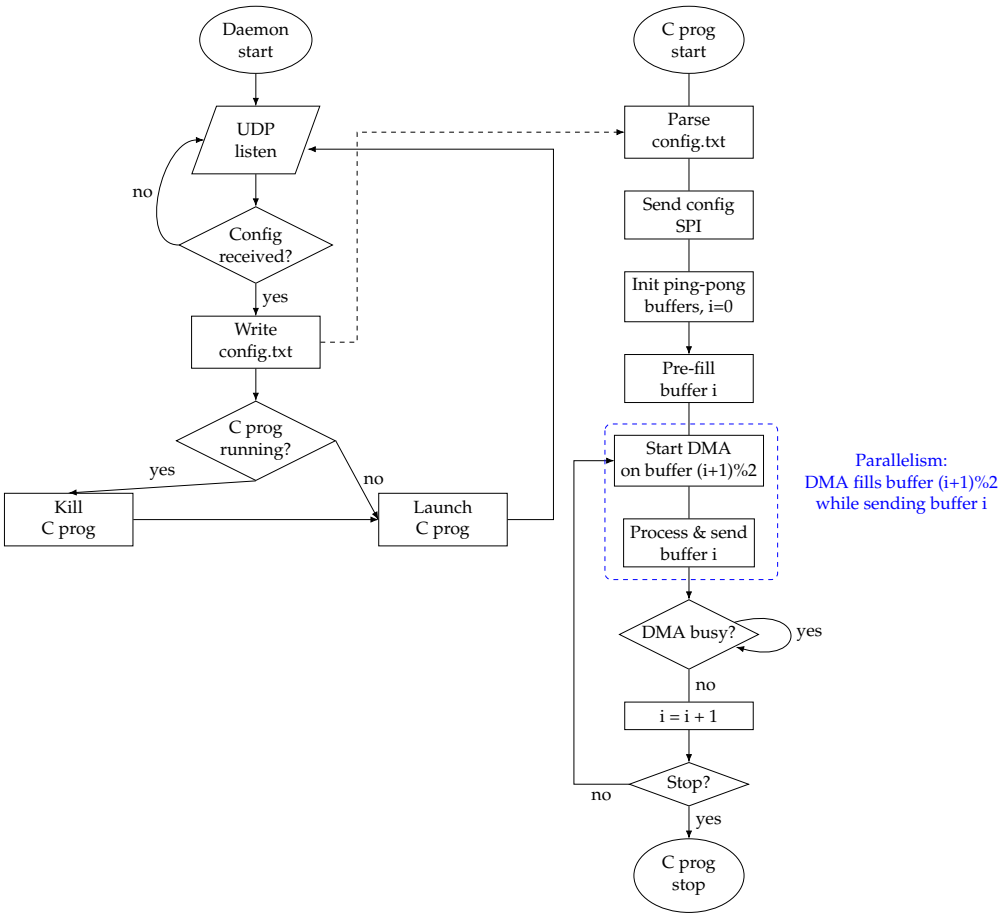

Figure 3.7 Flowchart of the software running on the HPS.

Programmable Logic (PL). The Cyclone V supports LVDS receivers; the 32-bit serial stream issued by the transceiver is captured at 64 MHz DDR. A dedicated finite-state machine (FSM) reassembles the incoming 2-bit slices into contiguous 32-bit words before they are pushed into a dual-clock FIFO that bridges the LVDS clock domain to the system clock. The FIFO feeds the DMA engine, which writes the words to DDR3 via the high-bandwidth F2SDRAM bridge, avoiding the lightweight bus except for configuration (Figure 3.8).

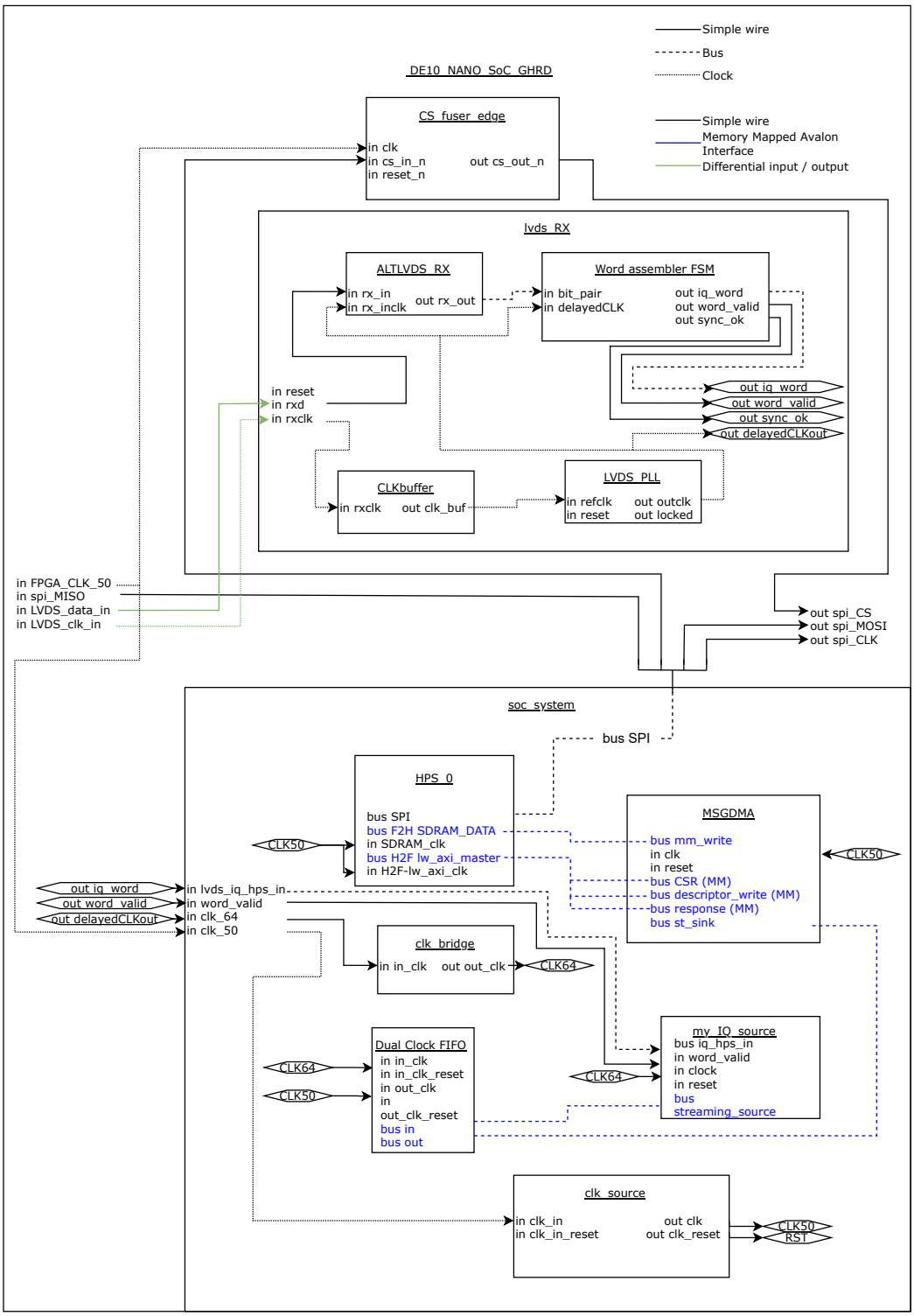

Figure 3.8 Cyclone V block diagram: instantiated HDL IPs and interfaces with the HPS.

A custom SPI *chip-select control* IP is also instantiated so that the PL can assert and release the chip-select line with cycle-accurate timing; its behavior is detailed in Sub-section 3.3.2.

PS/PL Interfaces.

- The **FPGA-to-HPS SDRAM (F2SDRAM) bridge** exposes the DDR3 controller directly to AXI masters instantiated in the PL, enabling high-throughput DMA without using the lightweight bridges.
- The **HPS-to-FPGA AXI Master** bridge allows the software to configure the DMA engine and to read its status registers through memory-mapped transactions.

3.3.2 SPI communication with the AT86RF215

SPI Controller

The built-in *DW-APB SPI* controller of the HPS is configured through the device-tree as `/soc/spi@0xfff00000` (see Subsection 3.3.7). In master mode it transmits fixed 8-bit words; once eight clocks have elapsed, the hardware releases the chip-select line (`SELN`) unconditionally before the next byte begins. This behavior violates the AT86RF215 timing requirements, which mandate that `SELN` stay low for the full command header and data payload [18].

To guarantee a continuous frame of *three bytes* (2-byte command + 1-byte data), the four SPI pins are exported to the FPGA, where a lightweight IP called `CS_fuser_edge` stretches the chip-select signal: This functionality is illustrated in Figure 3.9, which shows the correction performed by the `CS_fuser_edge` IP.

- **Inputs**: `clk`, `reset_n`, and the original `cs_in_n` from the HPS.
- **Parameter** `NB_SEG` sets the exact number of rising edges of `cs_in_n` (i.e. bytes) after which the fused `cs_out_n` is released. In practice `NB_SEG=3` for a 24-bit single-access.
- **Operation**: on the first falling edge of `cs_in_n`, `cs_out_n` is driven low and kept low until the counter reaches `NB_SEG`. Thereafter, it returns high exactly one cycle later, satisfying the slave's timing t_{SPI_9}.

This design preserves the Linux SPI driver (no kernel patches) and removes CS glitches, while allowing us to output the SPI on the GPIOs of the FPGA, avoiding the need for a separate SPI connector.

Project setting. The SPI clock is configured to 2 MHz in this implementation. It can be adjusted (within the transceiver datasheet limits) if required.

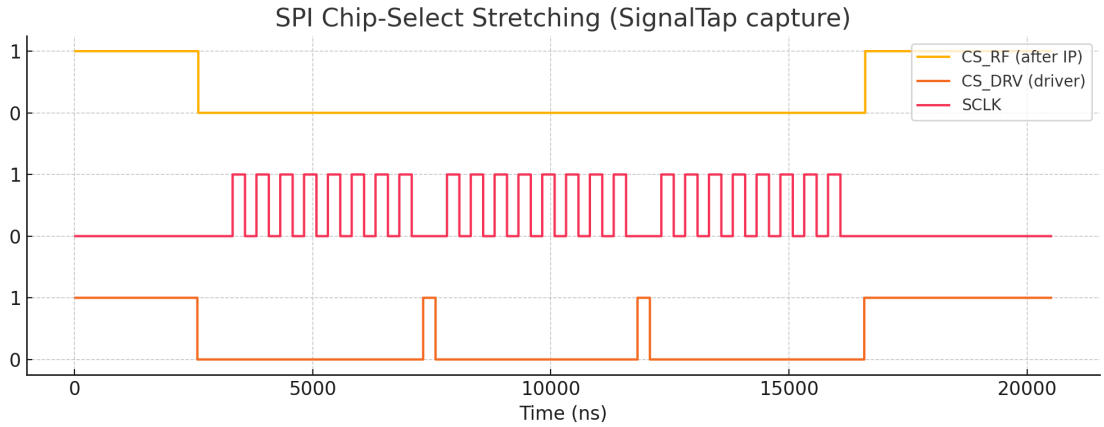

Figure 3.9 `CS_fuser_edge` IP operation.

Commands and Registers

The AT86RF215 transceiver configuration follows a structured initialization sequence implemented in the HPS software. After powering up, the system performs a complete device reset (RST register = 0x07) to ensure a clean starting state. The transceiver then requires a specific state transition sequence to enter receive mode: TRXOFF → TXPREP → RX, achieved through command register writes (CMD = 0x03 for TXPREP, CMD = 0x05 for RX).

The configuration parameters are loaded from an external file and written to the transceiver registers during initialization. Key parameters include the carrier frequency setup through the channel center frequency registers, receiver bandwidth configuration, digital front-end settings, and automatic gain control parameters. The I/Q interface is configured for LVDS output. This approach allows for flexible system reconfiguration without requiring firmware modifications, as all critical RF parameters can be adjusted through the configuration file loaded at runtime.

3.3.3 LVDS Data Reception and Handling in the FPGA

The reception chain for the I/Q samples is implemented by the `lvds_RX` IP core highlighted in Figure 3.8. It deserializes the LVDS stream provided by the AT86RF215. As mentioned in Subsection 3.2.2, the transceiver outputs a **64 MHz** DDR clock and a serial data stream that represents *32-bit words* at a fixed rate of 4 MS/s (128 Mbit/s).

Clock conditioning. The 64 MHz differential clock is forwarded from the AT86RF215 transceiver to the FPGA. This incoming clock first traverses a dedicated clock buffer, then a PLL (`LVDS_PLL`) that stabilizes and reproduces the 64 MHz domain used for all logic related to I/Q sample reception and processing. Since the clock-to-data skew can be trimmed directly inside the transceiver (register `IQIFC1.SKEWDRV`), the PLL phase-shift is left at zero and serves only for jitter filtering.

Serial-to-parallel conversion. An `ALTLVDS_RX` megafunction captures two bits per clock edge (serialization factor 2) and delivers a 2-bit vector every cycle. With a single data lane, this yields 32 LVDS bits per word.

Word framing state machine. A compact finite-state machine (FSM) inside `rf215_lvds_rx` assembles sixteen consecutive 2-bit slices into one 32-bit word while checking frame integrity:

- **HUNT** state searches the I_SYNC pattern `2'b10` in bits [31:30].
- Upon detection it moves to **SYNCED**. A 5-bit counter advances every cycle; after 16 cycles a complete word is available.
- If both sync fields of the word match `I_SYNC = 2'b10` and `Q_SYNC = 2'b01`, the FSM asserts `word_valid`; otherwise it flags a framing error and returns to **HUNT**.

The outputs are:

- `iq_word`: the 32-bit I/Q word, updated only when valid;
- `word_valid`: one-cycle pulse signalling a fresh word;
- `sync_ok`: high when frame alignment is locked, low for idle or error words.

The zero-words are ignored. They don't produce a `word_valid` signal and the output `iq_word` remains unchanged until a new valid word is received. It allows the IP to work for all sampling rates without requiring any change in the design.

Clock-domain crossing and DMA. Inside `soc_system`[5], the custom IP `my_IQ_source` forwards the validated 32-bit words over an Avalon-ST[6] source port to a dual-clock FIFO. This FIFO forms the boundary between the 64 MHz LVDS domain and the 50 MHz system domain of the FPGA and streams burst-aligned data to the DMA engine. Through the `F2SDRAM` bridge the DMA writes the samples into DDR3, where they are subsequently read by the HPS application for network transmission. This will be further detailed in Subsection 3.3.5.

The overall path therefore guarantees loss-free reception from the LVDS pins to system memory while respecting the timing rules stated by the transceiver datasheet.

3.3.4 Pin assignments and constraints

Cyclone V devices provide several independent I/O banks. Each bank is powered by its own `VCCIO` rail, so the designer is free to select the appropriate voltage per bank. Pins that share the same electrical standard must therefore be grouped inside a bank configured to the corresponding supply.

[5] *Platform Designer* (Intel Quartus Prime) integrates IPs and fabricates a system-on-chip netlist. The instance `soc_system` contains the HPS interfaces, DMA controller and memory interconnect.

[6] Avalon Streaming (Avalon-ST) is Intel's high-throughput, unframed data interface.

SPI interface (3.3 V LVCMOS). Bank 5 is unused elsewhere in the design; it is configured to 3.3 V and hosts all SPI signals. Because SPI is single-ended and comparatively slow, controlled-impedance routing is unnecessary; the intrinsic output resistance of the FPGA drivers suffices to damp residual reflections, and no on-chip termination is enabled.

LVDS interface (2.5 V differential). LVDS signals must be placed on a bank that (i) is set to 2.5 V `VCCIO` and (ii) offers dedicated differential pin pairs. Several banks fulfill these requirements; Bank 4A is selected here because it provides a convenient cluster of unused pairs. Those pairs come with valuable analog resources:

- factory-matched internal routing with less than 25 ps skew between the p and n lines,
- an integrated 100 Ω on-chip termination (OCT),
- a 1.2 V common-mode bias network,
- direct connectivity to the receiver SERDES.

Using arbitrary pins would forfeit these resources and jeopardize both the timing and the impedance budget required by the LVDS standard [25].

Summary of pin mapping

The FPGA pin assignments for both SPI and LVDS interfaces are organized according to the I/O bank constraints described above. The complete mapping of signals to specific FPGA pins, GPIO indices, I/O banks, and electrical standards is provided in Appendix B. A schematic view of the physical connections between the DE10-Nano and the transceiver is shown in Figure 3.10.

3.3.5 DMA Transfers Between the FPGA and the HPS

After reassembly, the 32-bit I/Q words together with the `word_valid` signal are exported from the `soc_system` block as a *conduit* interface[7]. A custom wrapper converts this conduit to an *Avalon-ST* stream in the LVDS domain (64 MHz).

[7] A conduit is a simple Platform Designer interface that carries application-specific signals without protocol overhead.

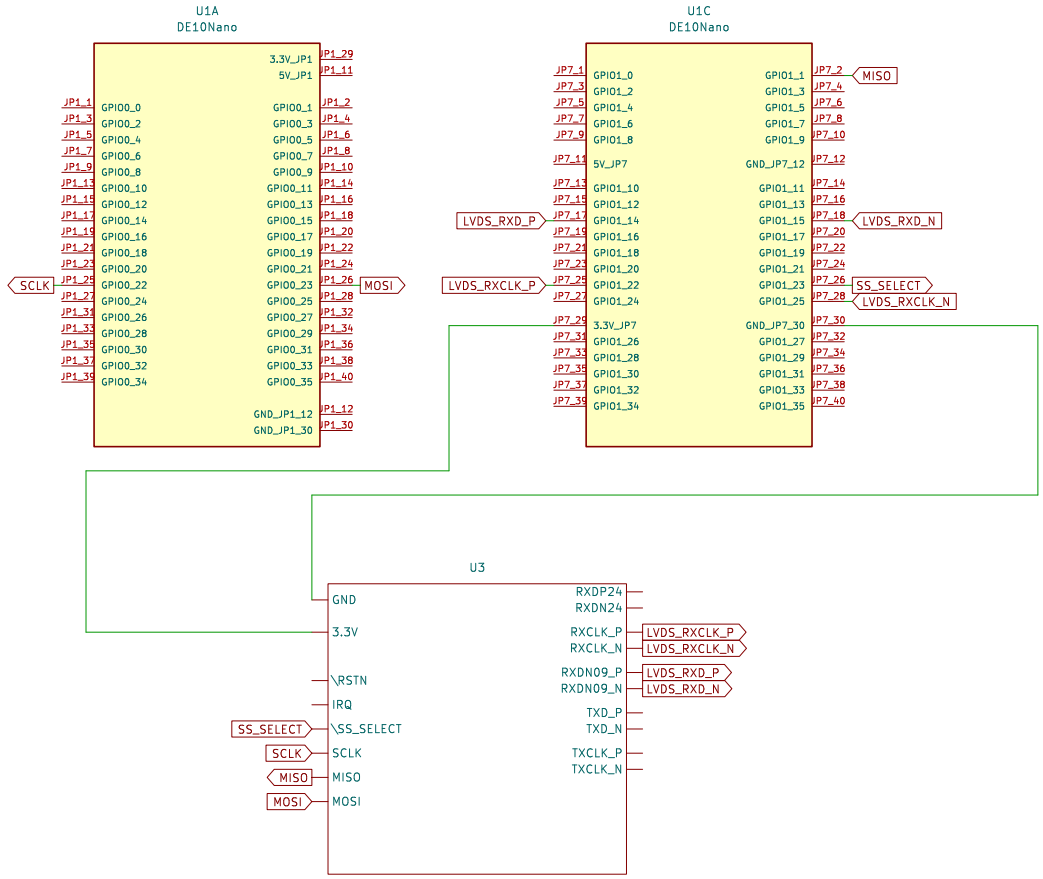

Figure 3.10 Pin connections between DE10-Nano and the AT86RF215.

Because the DMA and the memory subsystem operate at 50 MHz, the stream is first buffered by a dual-clock FIFO that bridges the 64 MHz LVDS clock to the 50 MHz system clock. Sampling clocks only determine when the FIFO checks for new data; they do not imply that fresh data appear on every cycle.

mSGDMA configuration. An *Intel Modular SGDMA* (mSGDMA) IP core is instantiated in Streaming-to-Memory-Mapped mode. Pairs of 32-bit samples are packed into a single 64-bit word before entering the DMA engine. The mSGDMA exposes five Avalon interfaces and one interrupt:

- `mm_write`—Avalon-MM *master*. Writes data to DDR3 through the `f2sdram` bridge.

- `st_sink`—Avalon-ST *sink*. Receives 64-bit sample words from the dual-clock FIFO.
- `csr`—Avalon-MM *slave*. Exposes control/status registers (CSR). The HPS polls the *Status* register to detect end-of-transfer.
- `descriptor_slave`—Avalon-MM *slave*. Accepts a four-word descriptor that sets the write address and transfer length, and launches the transfer when the *GO* bit is asserted.
- `response`—Avalon-MM *slave*. Returns post-transfer information such as actual bytes written and error codes.
- `irq`—Active-high interrupt. Unused in this work; the software operates in polling mode.

All base addresses given below belong to the FPGA (`soc_system`) address map. When accessed from the HPS they must be offset by the bridge window, as explained in Subsection 3.3.6. The detailed register maps for all mSGDMA interfaces are provided in Appendix D.3.

Status Register Details. Polling software uses the *Status* register to detect completion and fault conditions. Individual bits are defined in Table D.5 of the Appendix D.3. A transfer is finished successfully when `Busy` = 0 and `Stopped` = 1 while neither `StoppedOnError` nor `Resetting` is set.

3.3.6 HPS software and UDP transfers

The HPS software stack is designed to provide robust, high-performance streaming of I/Q samples from the FPGA to the host computer over UDP, with dynamic configuration and reliable resource management. The workflow is orchestrated by several components (Figure 3.7):

Python daemon and configuration listener. A custom Python daemon (`config_daemon.py`) is launched as a systemd service on the HPS. This daemon listens for incoming UDP packets on port 6000, which contain register configurations sent from the host (typically via GNU Radio or a dedicated sender block). Upon receiving a new configuration, the daemon writes the parameters to `config.txt` and (re)launches the C streaming executable. If a `STOP` command

is received, it terminates the running process. This mechanism allows remote, real-time reconfiguration of the transceiver and streaming parameters without manual intervention.

Systemd service for daemon launch. The configuration daemon is launched automatically at boot by a systemd service (see `config_daemon_transceiver.service`), which ensures reliable startup, restarts on failure, and proper integration with the system environment.

Accessing PL peripherals via memory-mapped bridges. To interact with peripherals instantiated in the Programmable Logic (PL), the HPS accesses their registers through memory-mapped bridges. The base addresses of these peripherals are defined in the FPGA address map, but when accessed from the HPS, an offset corresponding to the bridge window must be applied. The details of this address translation are provided in Appendix D.1.

High-priority launch script. To ensure maximum performance and minimal latency, the streaming executable is started via a shell script (`simple_launch.sh`) that sets CPU affinity, real-time scheduling, and disables process swapping. The script uses `taskset`, `chrt`, and `nice` to pin the process to a dedicated core and elevate its priority, ensuring uninterrupted DMA and UDP transmission even under heavy system load.

Python configuration sender. On the host side, a Python block generates the configuration file based on user parameters (frequency, chunk size, AGC/gain settings, etc.) and sends it via UDP to the HPS daemon. This block also sends a `STOP` command when the flowgraph is stopped, ensuring clean shutdown of the streaming process.

C streaming executable. The core of the data path is the `example_spi_dma` program, written in C. Upon launch, it loads the latest configuration from `config.txt`, initializes the SPI interface to the AT86RF215 transceiver, and configures all relevant registers (including AGC, frequency, and sampling rate). The program then sets up the DMA engine to transfer I/Q samples from DDR3 memory, packs the data into UDP chunks (each prefixed with a 64-bit sequence number for loss detection), and transmits them to the host computer. The transfer loop uses a ping-pong buffer strategy for continuous streaming, and supports

both test and UDP modes. Debug logging can be enabled to record raw I/Q samples to text files for analysis. Throughout this document the term *chunk size* denotes the number of payload bytes per DMA descriptor and per UDP datagram.

The installation, compilation, and configuration procedures for the daemon and program are detailed in Appendix F.2.

3.3.7 OS installation, configuration and bitstream loading

The Hard-Processor System (HPS) boots a full DEBIAN GNU/Linux distribution. Instead of building the whole software stack, the pre-built `Debian_for_DE10_Nano_v5.12.img.xz` image released by the *de10-nano* project (kernel 5.12, U-Boot 2021.07) was written to a 8 GiB micro-SD card with the *GNOME Disks* "DISC CREATOR" function under Ubuntu 24.04 LTS [26]. First boot required only Ethernet and the serial console; no further root-filesystem changes were necessary.

Device-tree customization. To enable communication between the HPS and the FPGA, several modifications to the device-tree are required:

- Activation of all HPS/FPGA bridges to expose the necessary memory-mapped interfaces.
- Reservation of a contiguous, uncached memory region for DMA operations.
- Addition of a user-accessible SPI master node for communication with the AT86RF215 transceiver.

These changes ensure proper integration of the hardware components with the HPS software stack. All device-tree excerpts and detailed steps for modification, recompilation, installation as well as the pre-configured SD-Card image are provided in Appendix C.1.

Uploading the Bitstream. Once the Quartus design is compiled, the bitstream must be converted to the Raw Binary File (.rbf) format and uploaded to the FAT partition of the micro-SD card. This ensures that the FPGA is programmed

with the correct bitstream during the boot process. The detailed procedure for bitstream conversion and upload is provided in Appendix C.2.

3.4 Host computer software interface

3.4.1 Host computer environment

The host computer used for development and testing is a Dell XPS 15 (x86 architecture) running Ubuntu 24.04 LTS 64-bit with GNU Radio version 3.10.9. The host computer connects to the DE10-Nano board via a standard Ethernet cable, enabling both SSH communication and UDP data transmission. In addition, the course-provided `gr_fsk` modules are installed to extend GNU Radio with custom FSK processing and utility blocks developed by the teaching team.

3.4.2 Network configuration

To enable SSH communication with the HPS and UDP data transmission between the DE10-Nano and the host computer, proper IP address configuration is required for both systems. This is achieved by configuring a DHCP server on the host computer, which automatically assigns an IP address to the HPS through the Ethernet connection. This network setup allows for seamless system control via SSH and real-time data streaming through UDP packets. The detailed DHCP server configuration is provided in Appendix E.

3.4.3 Software architecture and UDP data-stream reception

The receiver is entirely implemented in GNU RADIO, an open-source signal-processing framework that offers ready-made blocks for software-defined radio. The flowgraph comprises three functional tiers:

1. transmission of configuration parameters to the DE10-Nano,
2. real-time reception of IQ chunks over UDP,
3. basic baseband processing and demodulation.

Configuration forwarding

All run-time settings are exposed as GNU RADIO variables:

- `sample_rate_register_config` — selects the hardware sampling rate,
- `samp_rate` — automatically derived from the register value,
- `chunk_bytes` — payload size per DMA transaction *and* per UDP datagram,
- `AGC_ON` — enables or disables the automatic-gain control,
- `AGC_Target` — target power level when AGC is active,
- `manual_gain` — fixed RX gain when AGC is off,
- `center_freq_hz` — RF center frequency.

An embedded-Python block, `rf215_cfg_sender`, packs these parameters into a single UDP message and transmits it to the DE10-Nano on port 6000 at start-up. When the flowgraph terminates, the same block issues a final `stop` command to halt data acquisition on the board.

Remark on `chunk_bytes`: the default value is `1024`, i.e. a 1024-byte datagram. Changing it is possible, but because the datagram size is hard-coded in the Python helper, the script must be edited manually before running the flowgraph.

Reception of the data stream

UDP input. The built-in `UDP Source` block suffered from occasional sample loss, most likely due to buffer overruns. It was therefore replaced by a custom Python block, `udp_iq_vec_source_c`, which offers tighter buffer control and explicit sequence-number checking.

Operation of `udp_iq_vec_source_c`

- Listens on a user-defined port (default 5000) for datagrams consisting of an 8-byte sequence counter followed by raw IQ data.
- Validates the chunk length (`pkt_size = chunk_bytes + 8`); invalid datagrams are dropped.
- Detects lost chunks by monitoring the 64-bit counter and logs gaps.

- Converts each 32-bit word to two signed 13-bit integers, scales them to $[-1, 1]$, and forms `complex64` samples.
- Groups the samples into vectors of length `vec_len = chunk_bytes/4` and queues them for downstream processing; if the FIFO is full the vector is discarded (warning issued).
- Emits one output stream (no input) that can be wired directly into the remainder of the GNU RADIO graph.

Throttling. Because samples arrive in bursts, a `Throttle` block equal to the receiver sampling rate smooths the flow and protects subsequent stages from overrun.

Basic processing and preliminary demodulation

The IQ vectors are then passed to a chain identical to that of the previous platform:

- DC blocker,
- low-pass filter matching the useful bandwidth,
- preamble detector,
- timing-and-carrier synchronizer,
- demodulator.

The simplified flowchart is depicted in Figure 3.11.

3.5 Installation of Software Dependencies

Installation of the GNU Radio add-on `gr-fsk` and setup of the evaluation flowgraph are described in Appendix F.1. Configuration of the HPS daemon and compilation of the C program for the DE10-Nano are described in Appendices F.2 and F.2.2.

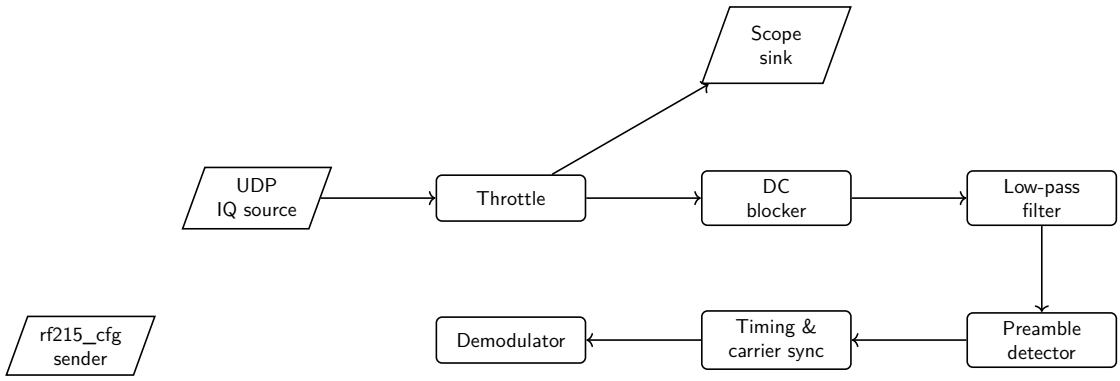

Figure 3.11 GNU Radio simplified flowchart.

4

Validation and Experimental Results

4.1 Validation methodology and setup description

This chapter details the methodology and the experimental setup used to verify the functionality and performance of the DE10-Nano–based receiver. The validation is organized in several sections, each addressing a specific stage of the signal path.

Although every block has been unit-tested, only certain interfaces offer a straightforward, quantitative check. For instance, UDP payloads can be compared bit-for-bit between the FPGA and the host computer. Conversely, other stages lack direct observation points or are time-aligned in hardware, making a one-to-one comparison impractical. A typical case is the LVDS link: the FPGA receives raw samples from the AT86RF215, yet the transceiver's internal FIFO prevents access to the exact sequence injected on the line.

For such stages, validation is performed indirectly. After demodulation on the host, a CRC is computed and verified. A zero error rate confirms that the samples have traversed the entire chain unaltered. In addition, the sampling clock is monitored at every boundary, providing an early warning if the stream slips or stalls.

The hardware arrangement mirrors the design introduced in Chapter 3. To

generate repeatable test traffic, a microcontroller unit (MCU) equipped with an RF front-end transmits framed packets through a shielded SMA cable to the AT86RF215 receiver. This cabled link eliminates over-the-air fading and guarantees a clean reference signal for the end-to-end tests.

The GNU Radio configuration settings used to validate the setup are listed in Table 4.1.

ID	Value	Correspondance
sample_rate_register_config	5	800 kS/s
center_freq_hz	868000000	868 MHz
chunk_bytes	1024	1024 bytes
AGC_on	0	manual gain
AGC_target[1]	3	-30 dBFS
manual_gain	6	18 dB
payload_len	100	100 bytes

Table 4.1 GNU Radio variables.

Most runtime parameters appear as GNU Radio variables (Table 4.1).
A small set of low-level RF filtering settings is still hard-coded in the HPS C initialization code (they could be exposed later but are not in the present implementation).
They cover: (i) the analog RX bandwidth and associated low-IF shift, and (ii) the digital post-filter cutoff (RCUT).
Table 4.2 summarizes the hard-coded fields used for the sub-GHz path (RF09).

Field	Value	Effective setting
RF09_RXBWC.IFS	1	IF shift enabled (moves low-IF away from DC)
RF09_RXBWC.BW	0x8	Analog low-IF BPF $\approx$ 1.0 MHz passband
RF09_RXDFE.RCUT	0x4	Post-filter cutoff $\approx$ 0.4 MHz (with 0.8 Msps)

Table 4.2 Hard-coded RF/filter register fields (currently not exposed to GNU Radio).

[1] In this configuration, the AGC is disabled, so this value is not used.

4.2 Validation of data streams and interfaces

4.2.1 SPI transfer integrity

The validity of the SPI transfer between the HPS and the AT86RF215 passing through the FPGA is verified at signals level first, using SignalTAP during a test sequence, checking manually the data extracted from the signals and finally comparing them to the value read by the HPS. The sequence test consists in reading the part number of the AT86RF215. As mentioned in the datasheet, the part number is 0x34 and is accessible in register `RF_PN` at address `0x000D` (Table A.1). The Figure 4.1 shows the signal measured by SignalTAP during the SPI read transaction. It shows a good functioning of the `CS_fuser_edge` IP. The manual decoding of the SPI transaction confirms that the address is correctly sent, and that the value is correctly transmitted by the radio chip.

The output of the HPS program confirms that this value is correctly recovered by the SPI driver:

```
Transceiver PN/VN = 0x34 / 0x03
```

4.2.2 LVDS data integrity between transceiver and FPGA

As explained in Section 4.1, it is not possible to probe the time-domain samples generated internally by the AT86RF215. A bit-for-bit comparison between transmitted and received data is therefore out of reach. The integrity of the LVDS link is instead verified by analyzing the timing of the `word_valid` strobes and the logical behavior of the decoded I and Q samples captured in the FPGA.

First, Figure 4.2 plots the positions of each `word_valid` event. With the SignalTAP logic analyzer clocked at 64 MHz and an expected I/Q-word rate of 800 kS/s, a new word must appear every 80 SignalTAP samples, that is every 1.28μs. The constant spacing visible in the plot confirms that this timing requirement is met over the entire capture.

Second, Figure 4.3 shows the decoded I and Q components when the transmitter is disabled. Both components stay within approximately ± 5 units of amplitude,

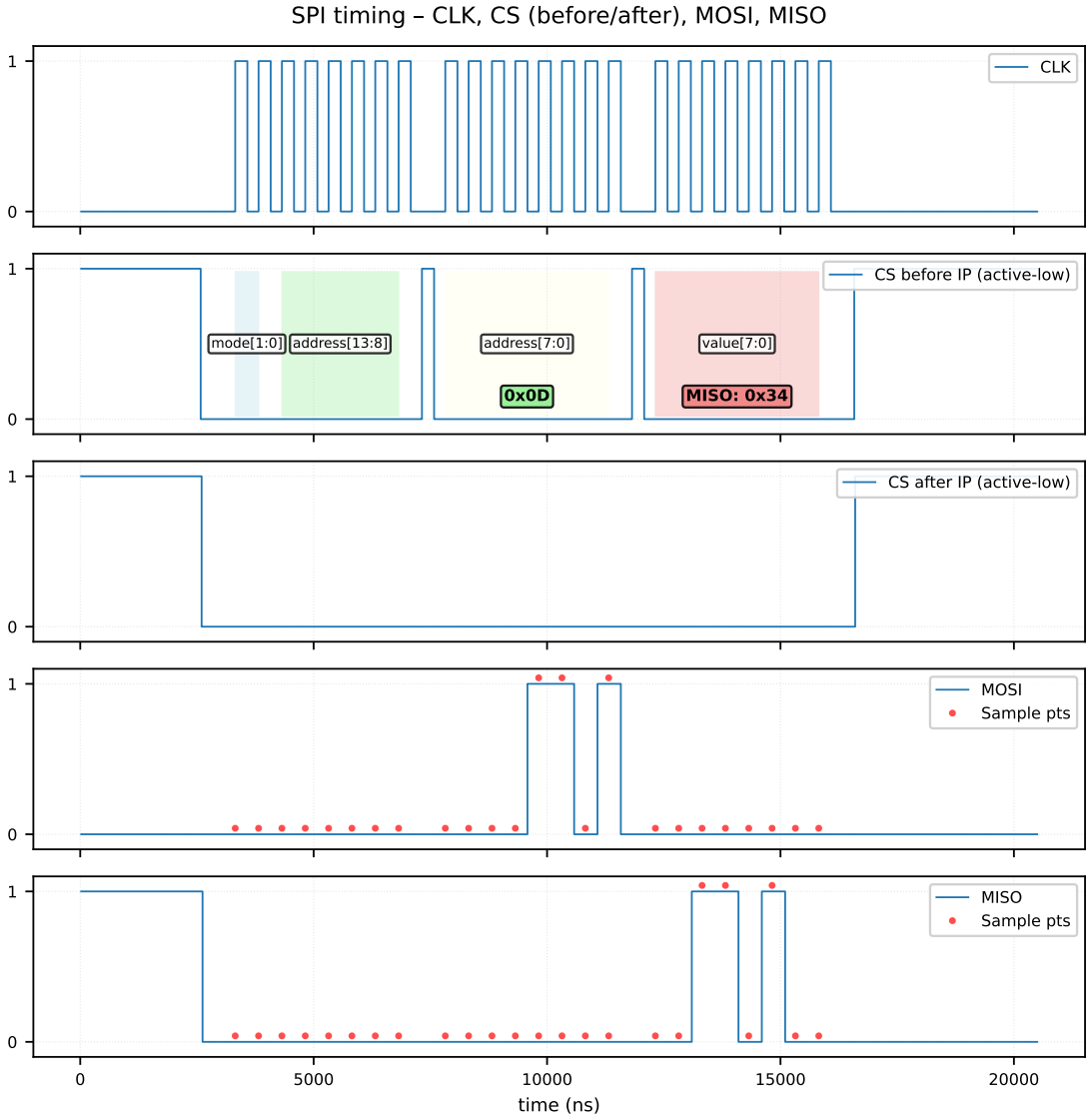

Figure 4.1 SPI read transaction (SignalTAP capture).

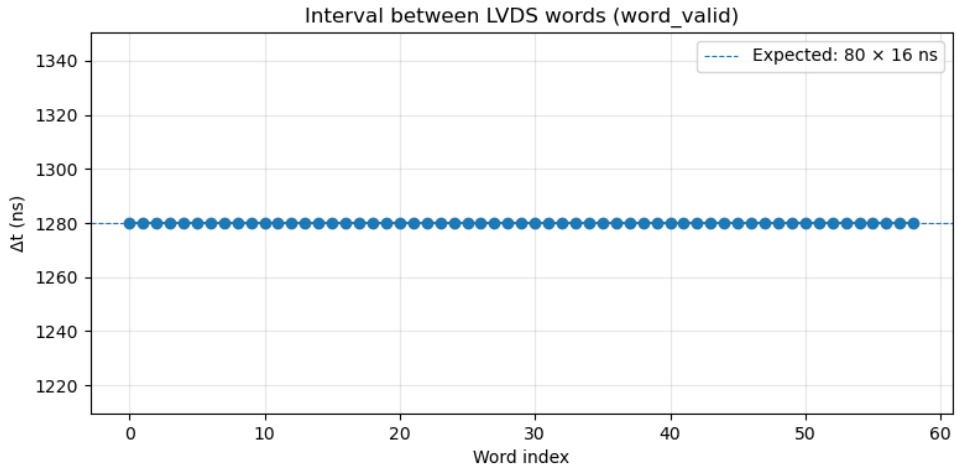

Figure 4.2 Interval between successive `word_valid` strobes extracted from SignalTAP.

indicating that only noise is present and that no spurious patterns are introduced by the interface.

Finally, when real data packets are transmitted, the captured samples (Figure 4.4) exhibit the expected characteristics of a GFSK signal: a continuous-phase waveform with a sinusoid-like shape, although the exact form cannot be predicted in advance. This provides a strong indication that the LVDS interface is functioning correctly and transporting the full-bandwidth signal without corruption, even if it does not constitute absolute proof.

4.2.3 FPGA to HPS DMA data transfer reliability

The reliability of the DMA transfer is assessed by comparing I/Q samples at three points along the chain: (i) the `LVDS_rx` IP output observed with SignalTAP, (ii) the `mm_write` bus driven by the DMA controller, and (iii) the samples read by the HPS from DDR and optionally dumped to a text log (Figure 4.4). This validates the custom data path end to end. The DMA controller is assumed reliable, as it is a standard SoC component validated in previous projects.

Because SignalTAP capture depth is limited, each acquisition window at the `LVDS_rx` output is short. This limits the number of samples that can be compared

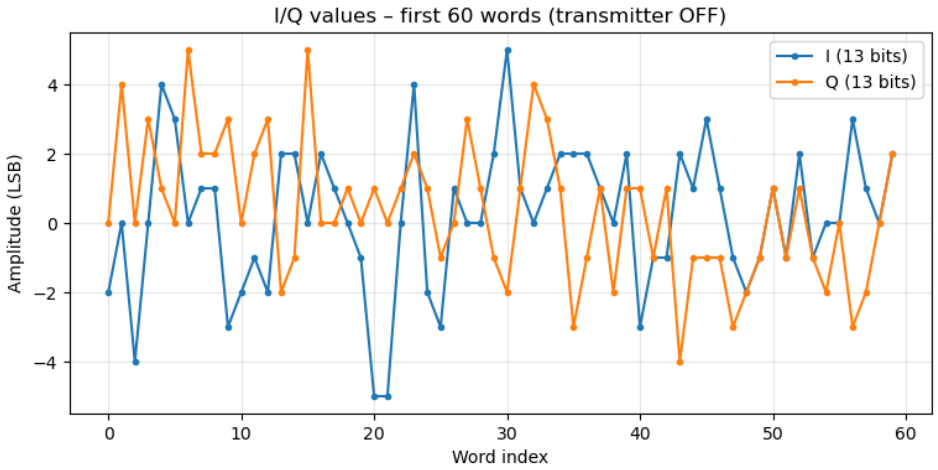

Figure 4.3 I and Q sample values with transmitter disabled (noise-only condition).

to those read by the HPS. However, it is sufficient to verify that the data path is functional.

The text log is generated by the HPS when the `DEBUG_TXT` option is enabled. It can operate in two modes: `DEBUG_MODE_TRIGGER` and `DEBUG_MODE_FULL`. The first mode only reads samples when a trigger condition is met (i.e. when the amplitude exceeds a threshold), while the second continuously reads all samples without filtering.

Indeed, in normal operation, the HPS loops fast enough on the DMA for it never to be idle. Most of the time, the FIFO doesn't even have the time to fill, and when a transfer is asked by the HPS, the DMA needs to wait for sample to fill the FIFO. This can be observed in Figure 4.6, where the DMA operates under an underload condition.

In opposition, the Figure 4.5 shows the SignalTAP capture when the HPS is not able to read the samples fast enough, causing the FIFO to fill and eventually overflow. It is not a problem itself that the FIFO fills up, it is designed to do so but in our case, the FIFO overflows because the HPS is not able to read the samples and initiate a new transfer fast enough.

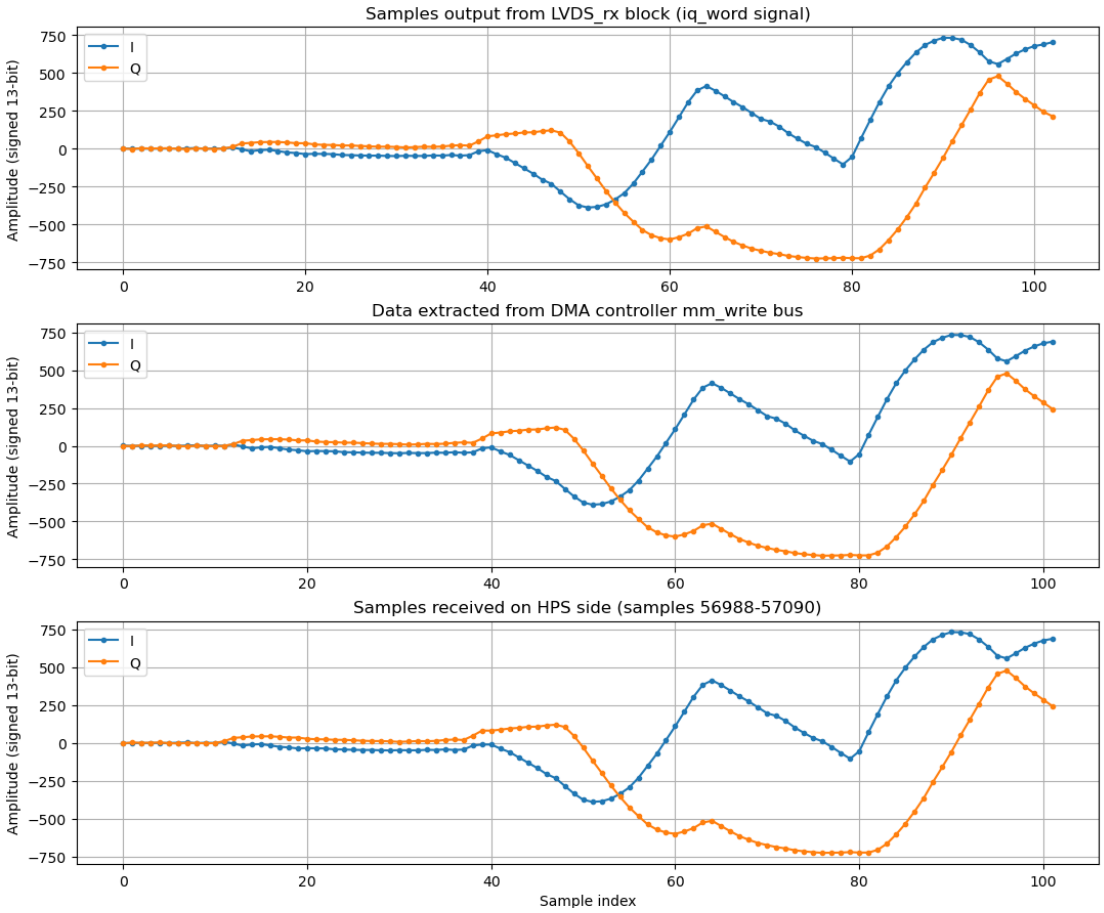

Figure 4.4 Comparison of the samples at the output of the `LVDS_rx`, at the `mm_write` bus of the DMA and read by the HPS from the DDR.

The reconstructed `wordI` and `wordQ` signals are visible in both DMA figures. They are extracted from the 64-bit `mm_write_writedata` bus.

Enabling `DEBUG_TXT` is purely a software action. It does not change the sampling rate at the `LVDS_rx` IP, which remains fixed at 800 kS/s. The bottleneck appears later: the HPS is slowed by file I/O, which delays DMA configuration and read-out. As a result, the DMA write FIFO fills and eventually overflows, causing sample loss on the HPS side.

When `DEBUG_TXT` is disabled, the measured rate is consistently 800 kS/s at all stages. However, when enabled, the rate processed by the HPS drops to $\approx$ 770 kS/s in `DEBUG_MODE_TRIGGER` and to $\approx$ 600 kS/s in `DEBUG_MODE_FULL`. This mode is used only for flow validation; it is not suitable for real-time streaming.

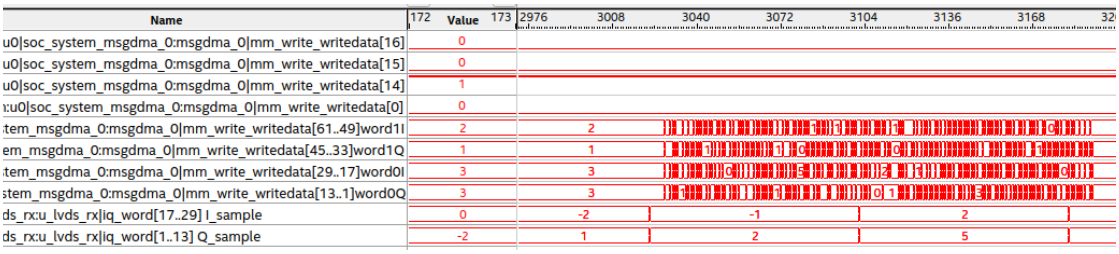

Figure 4.5 Overload conditions in the DMA transfer process, as observed in the Quartus SignalTap logic analyzer.

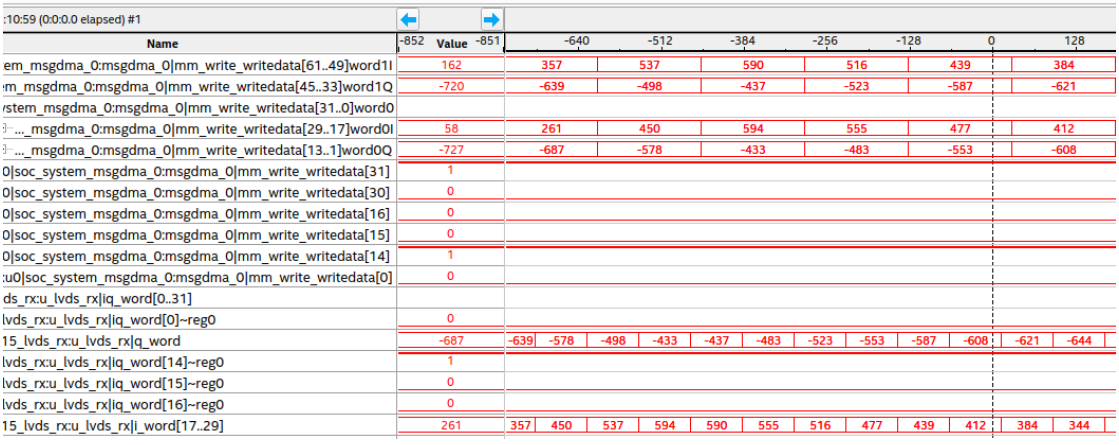

Figure 4.6 Normal operation of the DMA transfer, as observed in the Quartus SignalTap logic analyzer.

The DMA transfer size also impacts this behavior.
Increasing the chunk size (currently 1024 bytes) helps empty the FIFO in larger blocks and reduces the number of DMA triggers, but it must remain compatible with the network path constraints.

Classic IEEE 802.3 Ethernet limits the MAC client data ("Ethernet payload") of a non-jumbo frame to 1500 bytes [27]. Over IPv4 without options (20-byte IP header and 8-byte UDP header) the largest UDP payload that fits in a single Ethernet frame is $1500 - 20 - 8 = 1472$ bytes [28, 29].

The UDP protocol itself permits a much larger datagram because the 16-bit Length field allows up to 65 535 bytes total (header plus payload), giving a maximum IPv4 UDP payload of $65\,535 - 20 - 8 = 65\,507$ bytes when no IP options are present [29, 28].

Any payload exceeding the path MTU is fragmented at the IP layer, increasing overhead and the risk that loss of one fragment discards the entire datagram.
The chosen chunk size of 1024 bytes therefore remains a conservative value: it stays well below the typical 1472-byte non-fragmented UDP payload limit while still amortizing DMA descriptor and polling overhead.

4.2.4 UDP data reception and integrity on PC

The UDP reception on the host computer is validated by comparing the received samples with those sent by the HPS. To do so, a Python test script is used to retrieve the data from the UDP socket instead of following the usual reception process. This makes it possible to dump the received samples into a text file, which can then be compared with the samples sent by the HPS.

The sequence number is checked to ensure that no packets are lost during transmission. No packet loss was observed during the tests. The comparison between the sent and received packets is shown in Figure 4.7.

This test must also be performed with the `DEBUG_TXT` option enabled, as it is the only way to dump the samples sent by the HPS. However, the previously mentioned sample loss due to FIFO overflow when the HPS throttles does not

affect UDP reception, since the HPS is still able to send the samples it has read from the DMA. The only consequence is that the samples sent by the HPS are not all the samples captured by the FPGA, which does not pose a problem for validating UDP reception.

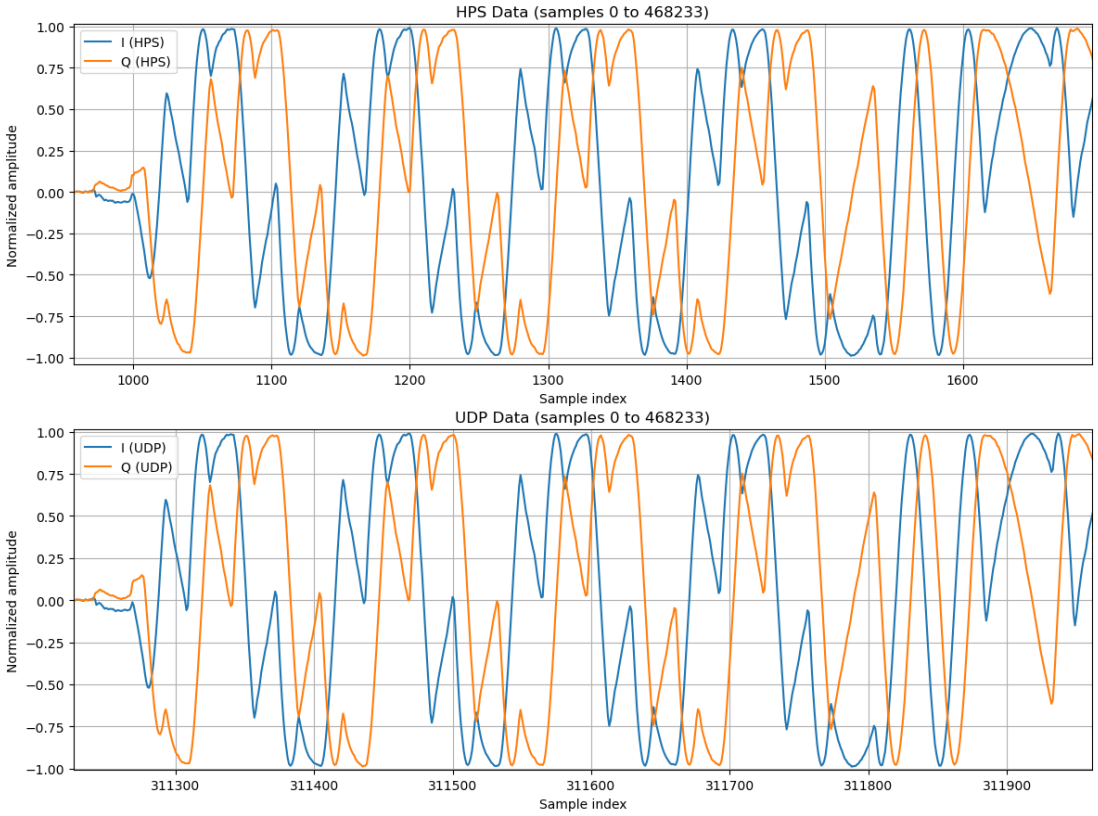

Figure 4.7 Comparison of the samples sent by the HPS and received by the PC.

In conclusion, the end to end UDP streaming path from the HPS to the host computer is validated.

4.3 Preliminary signal processing and demodulation tests

4.3.1 Demodulation tests in GNU Radio

The first step in validating the demodulation process is to use GNU Radio to process the I/Q samples captured from the AT86RF215. The I/Q samples are processed through the flowchart described in Subsection 3.4.3. The GUI of the `QT GUI Sink` block allows visual inspection of the I/Q samples in time and frequency domains, which is expected to exhibit a continuous-phase waveform characteristic of GFSK modulation. The activation of the demodulation and the detection threshold are also controlled by the `QT GUI Sink` block (Figure 4.8). As the threshold depends on the signal amplitude and therefore on the gain settings, it has been arbitrarily set to 0.01 as it seemed to be the value that allows demodulation in optimal conditions.

During the test where the transmitter sent packets over the GMSK modulation, the `QT GUI Sink` block displayed the expected continuous-phase waveform, confirming that the I/Q samples were correctly captured (Figure 4.8). The demodulated data stream was also successfully displayed in the GUI, indicating that the flowgraph was functioning as intended.

Over 100 packets, the demodulation process achieved a 100% success rate, with no errors or missed packets. This confirms that the demodulation process is robust and reliable, at least in the controlled environment of the experiment. Furthermore, it suggests that all stages of the data chain, from the AT86RF215 to the HPS, are functioning correctly, providing additional evidence for stages that could not be directly validated, such as the LVDS link.

The console output displayed after the last sample is shown below:

```
INFO:root:RX data rate : 800256 IQ S/s
INFO:sync:new preamble detected @ 1395509 (CFO 6558.76 Hz, STO 5)
INFO:sync:estimated SNR: inf dB (4096 samples, Esti. RX power: 3.15e-02,
     TX indicative Power: 0 dB)
INFO:parser:packet successfully demodulated: [ 0  1  2  3  4  5  6  7  8
 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99] (CRC: [195])
INFO:parser:100 packets received with 0 error(s)
```

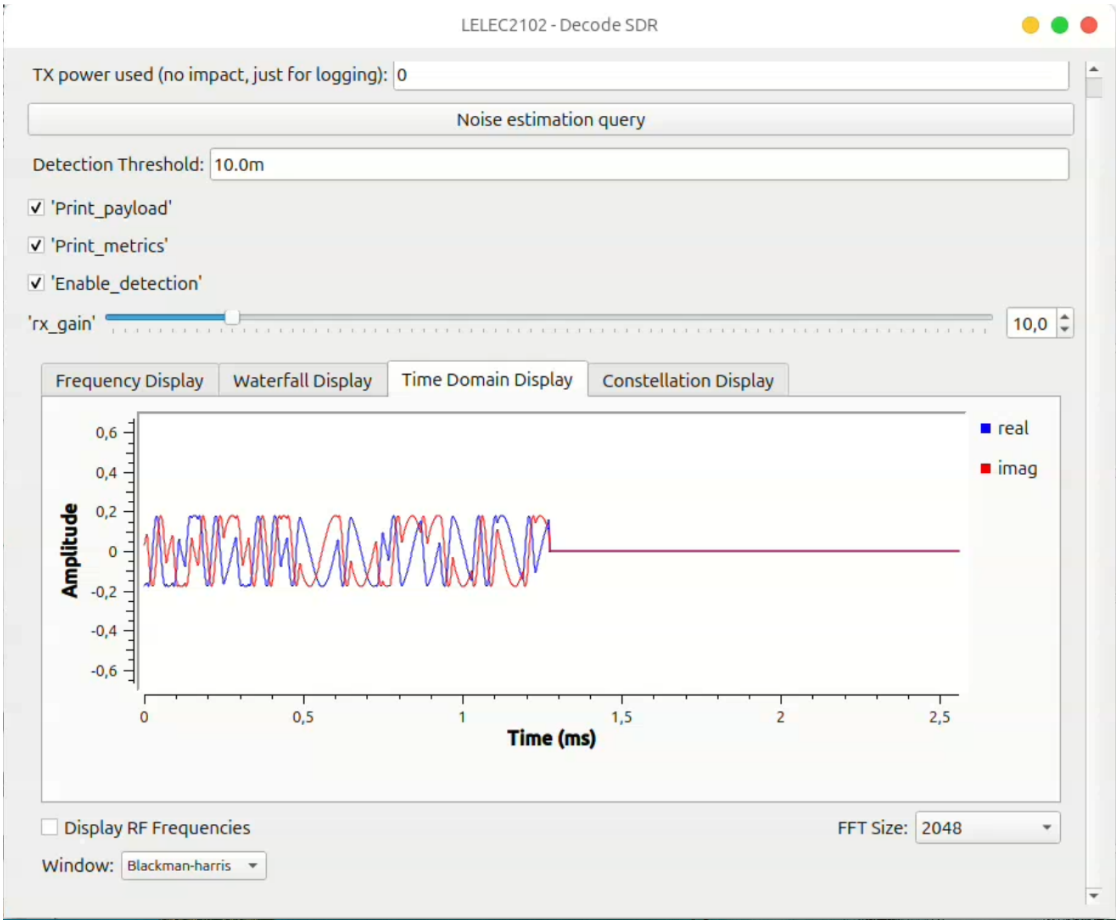

Figure 4.8 `QT Sink` block interface.

5

Alignment with the objectives, limitations, and perspectives

5.1 Alignment with the objectives

As a reminder, the initial objectives of this project were to:

1. Employ a widely used FPGA platform, already familiar to students and compatible with Quartus.
2. Fulfill the precise requirements of the ELEC Master's practical project.
3. Select an IQ transceiver module that is straightforward to implement and maintain.
4. Implement the complete data chain from antenna to host computer, passing through the radio transceiver and the FPGA.

FPGA Platform

In this work, we implemented a complete data chain using a DE10-Nano board. While this is not the FPGA platform most commonly used by students (the DE0-Nano is more common), it remains easy to use and compatible with Quartus. Apart from additional functionalities such as an integrated HPS and an external power supply, the DE10-Nano is very similar to the DE0-Nano and is available in large quantities at UCLouvain.

IQ Transceiver

The selected transceiver (AT86RF215) and its associated extension board (ATREB215-XPRO) are well documented. The transceiver's datasheet provides clear and detailed information on the module's specifications and usage. In addition, the PCB design files of the extension board are available, which facilitates integration into the overall system and provides a better understanding of its operation.

Although the PCB was not designed as part of this work, its cost is very reasonable (around €100 for the extension board including the embedded transceiver), and all signals are easily accessible, making it a suitable solution.

ELEC Master's practical project requirements and data chain implementation

The implemented data chain performs the same tasks as the original LimeSDR Mini setup, although configurable settings such as gain or carrier frequency differ slightly from the original platform. In addition, some parameter controls in GNU Radio are handled differently: instead of adjustable sliders (e.g., for RX gain), fixed variables are used for the duration of a running flowgraph. Nevertheless, the implemented chain is fully functional and enables reception of I/Q samples from the radio transceiver, which are then processed in GNU Radio successfully and without difficulty.

Due to time constraints, the embedded FIR filtering and energy detection functionalities were not implemented in the FPGA. In the previous receiver design, these features were necessary to handle issues with continuous data flow, which is why they were implemented in hardware. In our case, the UDP connection between the FPGA and the host computer allows reception of I/Q samples without any issues, even with a continuous flow of up to ≈ 1.5 MS/s. Moreover, the Low-Pass Filter (LPF) implemented in GNU Radio is sufficient to filter the received I/Q samples for the students' current needs.

Finally, the modular design of the receiver leaves the possibility to implement FIR filtering and energy detection in the FPGA in future work, if required.

5.2 Limitations of the current implementation

The current implementation of the receiver chain presents several limitations.

Limited data rate. The data rate is limited to 1.5 MS/s due to a bottleneck in HPS performance. This corresponds to a data flow of $\approx 1.5 \times 32 \times 10^6 =$ 48 Mbit/s for 32-bit samples. As the Ethernet link should theoretically reach 1 Gbit/s, we can assume that with better HPS performance it would be possible to achieve the maximum sampling rate of 4 MS/s. However, considering a useful bandwidth around 125 kHz, a sampling rate of 800 kS/s is already sufficient for the chosen demodulation scheme, leaving a comfortable margin. The current sampling rate limit is therefore not a bottleneck for this work.

AGC not implemented. Additionally, due to time constraints, the receiver chain with AGC enabled has not been implemented. While it is possible to activate AGC in the GNU Radio interface, its settings (AGCC.EN/FRZC for enable/freeze, AGCC.AVGS for the averaging window, AGCC.AGCI for the measurement point, AGCS.TGT for the target level,...) have not been tuned for this application, resulting in unusable signals and failed demodulation.

No FIR filtering or energy detection in FPGA. Moreover, as mentioned in Section 5.1, FIR filtering and energy detection features were not implemented in this design, even though they are not strictly required for this specific implementation.

Suboptimal wiring and board connections. Finally, the current experimental setup involves a significant number of wired connections. The transceiver and FPGA are mounted on separate boards, connected by individual wires. Due to constraints on voltage levels and LVDS pin assignments, these connections are distributed across the two GPIO headers of the DE10-Nano. The pin placement is likely suboptimal and could be improved in future revisions.

5.3 Perspectives for future work

In future work, several improvements could be made to the receiver design to fix its limitations and enhance its functionality:

Multithreading. Even though the current software implementation uses the ping-pong buffer architecture[1], it is currently implemented sequentially and is single-threaded. This means that the potential of the ping-pong buffer architecture is not fully exploited (Figure 3.7). The DMA transfer is initiated (buffer #2), then the previous samples (buffer #1) are read and sent over UDP. This process implies that the HPS must wait for the next transfer to be initiated before it can read and send the previous samples over UDP. Using two threads[2] could help to better utilize the available hardware resources and potentially increase the overall data throughput.

FIR filtering and energy detection in FPGA. Implementing the FIR filtering and energy detection functionalities in the FPGA would considerably lighten the load on the HPS, drastically reducing the amount of data sent over UDP, allowing for a higher sampling rate if needed, and improving the overall performance of the receiver. This would also allow for more complex filtering and detection algorithms to be implemented in hardware, which could further enhance the receiver's capabilities.

Improved I/Q dumping in the HPS. Currently, when the `DEBUG_TXT` option is enabled, the HPS writes each I/Q sample to a text file using standard file I/O calls in a tight loop. While this approach is simple, it introduces significant delays due to frequent system calls and text formatting operations, which can disrupt the real-time data stream. A more efficient method would be to implement buffered binary writes: samples would first be stored in a large memory buffer, and only flushed to disk periodically in large blocks. This reduces the number of write operations and avoids expensive text conversions, preserving more CPU cycles for real-time tasks and minimizing the impact on the data stream.

[1] The DMA fills up a buffer in the DDR memory, then the HPS reads it and sends the samples over UDP. The next DMA transfer writes in another buffer, allowing the HPS to read samples from the previous one without blocking.

[2] The HPS is a dual-core ARM Cortex-A9 processor, which can run multiple threads in parallel.

Improved hardware design. In addition to the software improvements, the hardware design could be improved to facilitate the integration of the transceiver and FPGA. If possible, using a single GPIO header instead of two, along with the use of a plug-in connector matching the GPIO header size with all the wires soldered to it, would greatly simplify the connections and make the design more robust. This would also allow for a more compact design, reducing the overall size of the receiver platform. Moreover, this connector could serve as a physical support for the transceiver, allowing it to be stacked directly on top of the DE10-Nano, making the design more compact and easier to handle.

6

Conclusion

This work set out to provide a practical replacement for the discontinued LimeSDR Mini while preserving access to raw baseband in the sub-GHz band and retaining the Quartus-based workflow used in the course. The proposed receiver platform meets that goal: it can be brought up quickly, operated reliably, and maintained.

End-to-end functionality was demonstrated on real captures. The transceiver is configured, I/Q samples are acquired consistently, and data are transported to the host reliably. The procedure is documented and ready for use in lab sessions, keeping the entry barrier low. A Debian image is provided to make the setup easily reproducible, and all source code and the Quartus project are included.

The design also fits cost and sustainability limits. One full station is about EUR 100, which is well below a typical LimeSDR replacement in the EUR 300 to EUR 400 range. The chosen transceiver has a clear and complete datasheet. This reduces integration risk and makes long-term support simpler.

Compared with the earlier LimeSDR-based design, the platform provides greater visibility across the receive chain and into the transceiver's settings. This clarifies how configuration choices affect the overall flow without changing the core architecture.

The scope is receive-only and sub-GHz. Within that frame, the main objectives

are reached: reuse of a known FPGA board, use of a simple I/Q transceiver, and a full chain from antenna to host. Validation so far used a direct SMA link between transmitter and receiver. Although several aspects remain to be improved and certain features could be added (for example, energy detection and multithreading), the objectives for this stage are met. The next step is to run over-the-air tests to check behavior in a realistic wireless setting and identify any additional calibration steps. The modular structure leaves room for future extensions.

Bibliography

[1] Jane Braxton Little. The worst wildfires are started by people. here's how. *Scientific American*, 2023. Accessed 2025-07-16.

[2] Sarita Bisht and Hukum Singh. *Advancing Forest Fire Management Practices: Policies and Strategies*, pages 279–293. Springer Nature Switzerland, Cham, 2025.

[3] Université catholique de Louvain. Lelec2103 - project in electrical engineering: Optimization of wireless embedded sensing systems, 2025. Accessed 2025-07-16.

[4] Lime Microsystems. Limesdr mini updates, 2022. Accessed 2025-07-16.

[5] Markus Dillinger, Kambiz Madani, and Nancy Alonistioti. *Software Defined Radio: Architectures, Systems and Functions*. John Wiley & Sons, 2003.

[6] Université catholique de Louvain. Lelec2102 - project in electrical engineering: Integration of wireless embedded sensing systems, 2025. Accessed 2025-08-15.

[7] Arun Paidimarri. *Circuits and Protocols for Low Duty-Cycle Wireless Systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge (MA), 2015. Available at: http://hdl.handle.net/1721.1/103674.

[8] Limesdr-mini v1.1 hardware description, 2019. Page last edited 2019-05-13 (oldid 2301).

[9] Lime Microsystems Ltd. *LMS7002M – FPRF MIMO Transceiver IC with Integrated Microcontroller*. Lime Microsystems, Guildford, United Kingdom, 2017. Document version 3.1r00.

[10] Intel Corporation, Santa Clara, CA. *Intel MAX 10 FPGA Device Datasheet*, 2022. Document ID 683794, current revision.

[11] Lime Microsystems Ltd., Guildford, United Kingdom. *LimeSDR-Mini v1.2 Hardware Description*, n.d. No date; consulted 2025-07-27.

[12] Terasic Technologies Inc., Hsinchu, Taiwan. *DE10-Nano User Manual*, 2016. Revision 1.0.

[13] Terasic Technologies Inc., Hsinchu, Taiwan. *DE0-Nano FPGA Development Kit – Board User Manual*, 2013. Revision 1.1.

[14] Intel Corporation, Santa Clara, CA. *Cyclone V Device Handbook, Volume 1: Device Overview*, 2023. Device family 5CSEBA6, current revision.

[15] Intel Corporation. *MAX 10 Embedded Multipliers User Guide*, 2024. Document ID 683467, current revision.

[16] Analog Devices Inc. *AD9361 RF Agile Transceiver Datasheet*, 2024. Revision G.

[17] Analog Devices Inc. *AD9363 RF Agile Transceiver Datasheet*, 2016. Revision D.

[18] Microchip Technology Inc. *AT86RF215/AT86RF215IQ Dual-Band Sub-1 GHz/2.4 GHz Transceiver Datasheet*, 2016. Document 42415.

[19] Microchip Technology Inc. *ATREB215-XPRO Extension Board User Guide*, 2017.

[20] Microchip Technology Inc. *REB215-XPRO-A PCB Specification*, 2017.

[21] Zangman. Debian for de10-nano (cyclone v soc), 2020. Accessed 2025-07-31.

[22] Telecommunications Industry Association. Ansi/tia/eia-644-a: Electrical characteristics of low-voltage differential signaling (LVDS) interface circuits, 2001.

[23] National Semiconductor Corporation. Lvds overview and system noise budgeting. Technical Report Application Note 903, National Semiconductor, 1998.

[24] Reb215_xpro_a design documentation. Design Documentation A09-2353, Atmel R&D India, Kandanchavadi, Chennai, 2021. Includes schematic/assembly prints and BOM (REB215_XPRO_A_TopLevel.SchDoc, REB215_XPRO_A_Connector.SchDoc, REB215_XPRO_A_RF.SchDoc).

[25] Altera Corporation, San Jose, CA. *Pin Information for the Cyclone® V SE 5CSEBA6 Device*, version 1.5 edition, December 2016. Document rev. 1.5, released 23 Dec 2016.

[26] Zolt Angman. Debian for de10-nano v5.12 (release notes), 2021. Accessed 2025-08-05.

[27] Charles Hornig. A standard for the transmission of ip datagrams over ethernet networks. RFC 894, Internet Engineering Task Force, April 1984.

[28] J. Postel. Internet protocol. RFC 791, Internet Engineering Task Force, September 1981.

[29] J. Postel. User datagram protocol. RFC 768, Internet Engineering Task Force, August 1980.

[30] Intel Corporation. *Modular Scatter-Gather DMA (mSGDMA) Intel® FPGA IP User Guide*, 2023. Document ID UG-01086.

A

AT86RF215 States and Registers

A.1 Finite State Machine for the AT86RF215 Transceiver

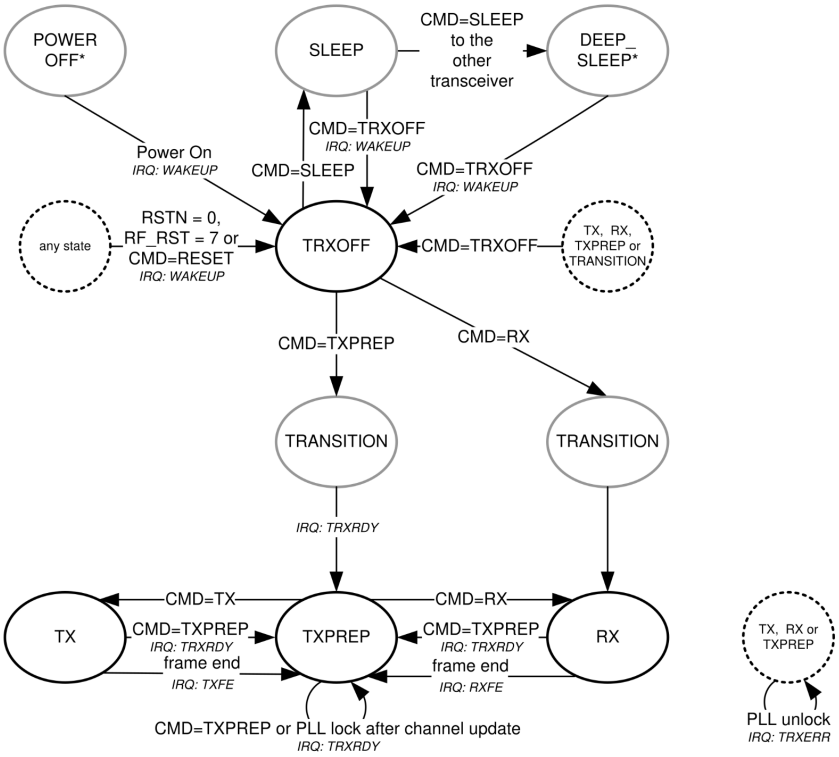

Figure A.1 Finite State Machine for the AT86RF215 Transceiver.

Source: AT86RF215 Datasheet [18]

A.2 AT86RF215 Configuration Registers

Register	Address	Function
RF09_IRQS	0x0000	RF09 interrupt status register
RF_RST	0x0005	Full transceiver reset (0x07 to reset RF09, RF24, and baseband)
RF_IQIFC0	0x000A	I/Q data interface configuration: LVDS IEEE mode, common mode voltage 1.2V (CMV1V2)
RF_IQIFC1	0x000B	LVDS mode configuration and clock-data skew delay (SKEWDRV, CMV1V2)
RF_PN	0x000D	Part number identification register
RF_VN	0x000E	Version number identification register
RF09_STATE	0x0102	RF09 transceiver state register
RF09_CMD	0x0103	RF09 state machine: TXPREP, RX, etc.
RF09_CS	0x0104	Enables I/Q streaming mode and sets external clock selection
RF09_CCF0L	0x0105	Receiver carrier frequency (LSB)
RF09_CCF0H	0x0106	Receiver carrier frequency (MSB)
RF09_CNL	0x0107	Channel number parameters (LSB, affects internal tuning)
RF09_CNM	0x0108	Channel number parameters (MSB, affects internal tuning)
RF09_RXBWC	0x0109	Analog low-pass filter bandwidth selection
RF09_RXDFE	0x010A	Digital RX configuration: sample rate, decimation, post-filtering
RF09_AGCC	0x010B	Automatic Gain Control configuration: on/off, target level
RF09_AGCS	0x010C	Automatic Gain Control status and manual gain setting

Table A.1 Configuration registers used in this project (sub-GHz, I/Q streaming mode).

Source: AT86RF215 Datasheet [18]

B

FPGA Pin Assignments

The FPGA pin assignments must be properly configured in Intel Quartus Prime to ensure correct functionality and electrical compatibility. This configuration process involves two main steps:

- **Pin Planner**: Physical pin assignments are defined in the Pin Planner tool, which allows mapping of logical signal names to specific FPGA pins. This ensures that each interface signal is routed to the appropriate physical pin on the device.
- **Assignment Editor**: Electrical constraints and properties are configured in the Assignment Editor, including I/O standards, drive strength, slew rate control, and termination resistances. For LVDS signals, the 100Ω on-chip termination (OCT) is enabled to match the differential impedance requirements.

GitHub Repository

The complete Quartus Prime project with all pin assignments and constraints is available in the GitHub repository: `https://github.com/Qwaks913/SDR_DE10_AT86RF215`

The complete pin mapping and electrical configuration for this project is summarized in the following table:

Intf	Signal	FPGA pin	GPIO index	Bank	I/O std.	Notes
SPI	`CLK`	Y19	GPIO_0[22]	5A	3.3-V LVCMOS	–
	`CS`	AD26	GPIO_1[3]	5A	3.3-V LVCMOS	–
	`MISO`	AC24	GPIO_1[1]	5A	3.3-V LVCMOS	–
	`MOSI`	AB23	GPIO_0[23]	5A	3.3-V LVCMOS	–
LVDS	`CLK(p)`	AF22	GPIO_1[22]	4A	LVDS	100 Ω OCT
	`CLK(n)`	AF21	GPIO_1[25]	4A	LVDS	100 Ω OCT
	`DATA(p)`	AG23	GPIO_1[14]	4A	LVDS	100 Ω OCT
	`DATA(n)`	AF23	GPIO_1[15]	4A	LVDS	100 Ω OCT

Table B.1 FPGA pin assignments with corresponding GPIO indices.

C

Configuration and setup of the DE10-Nano Board

C.1 Device-tree Modifications and Installation

This appendix provides the detailed steps and code excerpts for modifying, recompiling, and installing the device-tree on the DE10-Nano board.

GitHub Repository

A pre-configured SD card image with all necessary device tree modifications is available at: https://github.com/Qwaks913/SDR_DE10_AT86RF215

C.1.1 Required Modifications

- **Enabled FPGA bridges:** All four HPS / FPGA bridges (`lwhps2fpga`, `hps2fpga`, `fpga2hps`, `fpga2sdram`) must be set to `status = "okay"` and the property `bridge-enable = <1>;` added. This exposes the Lightweight, High-performance, and SDRAM paths as soon as the `altr,socfpga-fpga-bridge` driver is probed.
- **Reserved DMA buffer:** A contiguous, uncached 16 MiB block beginning at `0x0100_0000` is declared in `/reserved-memory`. The `no-map` flag prevents

kernel mapping, ensuring the region is safe for DMA traffic and user-space `mmap()`:

Device Tree: Reserved Memory

```
/memreserve/ 0x0000000001000000 0x0000000001000000;

reserved-memory {
   #address-cells = <1>;
   #size-cells      = <1>;
   ranges;
   dma_buffer:  dma_buffer@01000000 {
         reg      = <0x01000000 0x01000000>; /* 16 MiB */
         no-map;                              /*
uncached */
   };
};
```

- **User-accessible SPI master:** Activate the existing DesignWare APB SSI controller node (its `status` was changed from `"disabled"` to `"okay"`) *and add* a new child node `spidev@0` so chip-select 0 is bound to the generic `spidev` driver for user-space access (the child node was not present in the vendor device-tree). The resulting fragment is:

Device Tree: SPI Configuration

```
spi@fff00000 {
   compatible                = "snps,dw-apb-ssi";
   #address-cells      = <1>;
   #size-cells          = <0>;
   reg                        = <0xfff00000 0x1000>;
   interrupts              = <0 0x9a 0x4>;
   num-cs                    = <1>;
   clocks                    = <0x32>;
   resets                    = <0x06 0x32>;
   reset-names           = "spi";
   status                    = "okay";

   /* Added child node to expose CS0 as a
      raw SPI device to user space */
   spidev@0 {
         compatible               = "spidev";
         reg                        = <0>;
         spi-max-frequency = <25000000>;
   };
};
```

C.1.2 Recompiling and Installing the Device-tree

1. Install the device-tree compiler:

   Installation Command

   ```
   sudo apt install device-tree-compiler
   ```

2. Mount the FAT partition that stores the DTB:

   Mount Commands

   ```
   sudo mkdir -p /mnt/fat
   sudo mount /dev/mmcblk0p1 /mnt/fat
   ```

3. Apply the modifications specified above and compile the `*.dts` file into a

`*.dtb` binary:

Compilation Command

```
dtc -I dts -O dtb -o socfpga_cyclone5_de0_nano_soc.dtb
socfpga.dts
```

4. Copy the new DTB in place of the factory file:

Copy Command

```
sudo cp socfpga_cyclone5_de0_nano_soc.dtb /mnt/fat
```

5. Reboot the board to apply the changes.

C.1.3 Verification

After rebooting, verify the changes:

Verification Commands

```
cat /sys/class/fpga_bridge/*/state
ls -l /dev/spidev0.0
grep dma_buffer /proc/iomem
```

All four bridges should report *enabled*, the SPI master should be available to user-space, and the DMA buffer should appear as a reserved, uncached region.

C.2 Bitstream Conversion and Upload Procedure

This appendix details the steps required to convert the Quartus bitstream to the Raw Binary File (.rbf) format and upload it to the DE10-Nano board for automatic FPGA configuration at boot.

GitHub Repository

The complete SD card image with pre-configured device tree and bitstream is available in the `Releases` section at: https://github.com/Qwaks913/SDR_DE10_AT86RF215

C.2.1 Bitstream Conversion

1. Open Quartus and navigate to `File > Convert Programming Files`.
2. In the dialog box, select the format `.rbf` (Raw Binary File).
3. Name the file `soc_system.rbf`.
4. Choose the mode `Passive Serial x16`.
5. Complete the conversion process to generate the `soc_system.rbf` file.

C.2.2 Uploading the Bitstream

1. Mount the FAT partition of the micro-SD card on the DE10-Nano:

   Shell Commands

   ```
   sudo mkdir -p /mnt/fat
   sudo mount /dev/mmcblk0p1 /mnt/fat
   ```

2. Copy the generated `soc_system.rbf` file to the FAT partition:

   Shell Command

   ```
   sudo cp soc_system.rbf /mnt/fat
   ```

3. Reboot the board. The FPGA will be automatically configured with the new bitstream at startup.

C.2.3 Notes

- Ensure the file is named exactly `soc_system.rbf` and placed in the correct location on the FAT partition. - If multiple designs are used, update the bootloader or configuration scripts as needed. - The MSEL configuration pins (mode select DIP

switches) on the DE10-Nano must all be set to ON (Passive Serial x16) so that the bootloader can configure the FPGA with the `soc_system.rbf` at startup.

D

Memory-Mapped transactions between the HPS and the FPGA

D.1 Accessing the FPGA peripherals from the HPS

D.1.1 Local addresses of the FPGA peripherals

The FPGA peripherals are mapped with a local address space relative to each address domain. An address domain describes the range of addresses that can be accessed by a specific bus or interface. The `h2f_lw_axi` constitute an address domain, for which the FPGA peripherals are mapped to the following local addresses:

Component	Interface	Base	End	Size (bytes)
`msgdma_0`	`csr`	`0x0000_0000`	`0x0000_001f`	32
`msgdma_0`	`descriptor_slave`	`0x0000_0020`	`0x0000_002f`	16
`msgdma_0`	`response`	`0x0000_0040`	`0x0000_0047`	8

Table D.1 Local address map of `msgdma_0` inside `soc_system` (relative to the interconnect window).

D.1.2 Offset of the FPGA peripherals in the HPS memory map

These peripherals are accessible through the bridges enabled in the device tree of the HPS. Each bridge has a specific offset that is added to the local address of the peripheral to obtain the final address in the HPS memory map. The offsets for the bridges are as follows (Table D.2): All the FPGA peripherals are mapped

Bridge	Data width	Start (HPS phys)	End	Window size
h2f_lw_axi	32-bit fixed	0xFF200000	0xFF3FFFFF	2 MB
h2f_axi	32/64/128-bit	0xC0000000	0xFBFFFFFF	960 MB

Table D.2 Base address ranges of the HPS-to-FPGA bridges (compute: base + Qsys offset).

Source: [12, 21]

in the `h2f_lw_axi` bridge, which is a 32-bit fixed-width bus. The `h2f_axi` bridge is not used in this project.

D.2 Accessing the DDR3 memory from the FPGA

The DDR3 memory is accessible from the FPGA through the `F2SDRAM` bridge. As the DMA is only used to transfer data to the DDR3 memory, the latter is the only peripheral on that address domain for this bus and there is no need to add an offset bridge. The DDR3 memory is then accessible without any offset from the FPGA, with the same raw addresses as the ones mapped in the HPS.

In this implementation, the FPGA specifically accesses a reserved buffer region starting at address `0x01000000` and extending to `0x01FFFFFF`, providing a 16 MB dedicated buffer space for I/Q sample storage. This buffer allocation ensures that DMA transfers from the FPGA do not interfere with other system memory usage while providing sufficient capacity for high-throughput data streaming operations.

D.3 mSGDMA Register Maps

D.3.1 Descriptor Slave Port

Offset	Access	Field (bits)
0x00	Write	Read Address [31:0]
0x04	Write	Write Address [31:0]
0x08	Write	Length [31:0] (number of bytes)
0x0C	Write	Control [31:0]—*GO* = bit 0

Table D.3 Standard descriptor format written by the HPS to `descriptor_slave`.

Source: [30]

D.3.2 Control/Status Registers (CSR)

Offset	Attribute	Register
0x00	Read/Clear	Status
0x04	Read/Write	Control
0x08	Read	Write Fill Level [15:0] / Read Fill Level [15:0]
0x0C	Read	*Reserved* / Response Fill Level [15:0]
0x10	Read	Write Seq. Number [15:0] / Read Seq. Number [15:0]
0x14	Read	Component Configuration 1
0x18	Read	Component Configuration 2
0x1C	Read	*Reserved* / Component Version [7:0]

Table D.4 mSGDMA CSR map (`csr` interface).

Source: [30]

Bit	Name	Meaning
9	IRQ	Latched when any enabled interrupt event occurs. Cleared by writing a '1'.
8	Stopped on Early Term.	Dispatcher halted after encountering EOP before expected length.
7	Stopped on Error	Dispatcher halted due to error condition (see Response Port).
6	Resetting	Set during software reset; clears when reset sequence completes.
5	Stopped	Set after graceful stop or when `Stopped on Error/Early Term.` is set.
4	Response Buffer Full	Internal response FIFO is full.
3	Response Buffer Empty	Internal response FIFO is empty.
2	Descriptor Buffer Full	Descriptor FIFO is full.
1	Descriptor Buffer Empty	Descriptor FIFO is empty.
0	Busy	Dispatcher holds commands or a host port is still active.

Table D.5 Status register bit definition.

Source: [30]

D.3.3 Response Port

ST Data Bits	Access	Field (bits)
[31:0]	RO	Actual bytes transferred [31:0]
[39:32]	RO	Error [7:0]
40	RO	Early termination
41	RO	Transfer complete IRQ mask
[49:42]	RO	Error IRQ mask
50	RO	Early termination IRQ mask
51	RO	Descriptor buffer full
[255:52]	RO	Reserved

Table D.6 mSGDMA response source port bit fields.

Source: [30]

E

DHCP server configuration on the host computer

In order to allow the DE10-Nano to receive a valid IP address and access the internet through the host computer, a DHCP server was configured using the `NetworkManager` service under Ubuntu 24.04 LTS.
Rather than setting up `dnsmasq` or `isc-dhcp-server` manually, a shared connection profile was created for the Ethernet interface `enx00e0222d9e23` using the following command:

NetworkManager Configuration

```
nmcli connection add type ethernet ifname enx00e0222d9e23 mode
\
      shared con-name partage-de10
```

This configuration enables the following features:

- A static IP address (10.42.0.1/24) is assigned to the host interface;
- A DHCP server (based on `dnsmasq`) automatically allocates IP addresses in the subnet to connected devices;
- NAT and IP forwarding are handled by `NetworkManager`, allowing internet access from the DE10-Nano via the host.

The configuration of the connection `partage-de10` confirms that the `ipv4.method` is set to `shared`, which activates all aforementioned features. Once activated with:

Connection Activation

```
nmcli connection up partage-de10
```

the system behaves as a DHCP server and gateway for the DE10-Nano connected through Ethernet.

F

Installation of Software Dependencies

F.1 Installing the `gr-fsk` GNU Radio module

Prerequisite

GNU Radio must already be installed and working on the target computer. The `gr-fsk` module is provided by the teaching staff of the UCLouvain project courses LELEC2102 and LELEC2103 [6, 3].

F.1.1 Build and install

Open a terminal, change directory to the project folder that contains the `gr-fsk` module (e.g., `telecom/hands_on_wireless_measurements/gr-fsk`), then run:

Build Commands

```
mkdir build
cd build/
cmake ..
make install
ldconfig
```

F.1.2 Ensure the module is discoverable by GNU Radio

During installation, the console log prints paths similar to:

Example Output

```
/usr/local/XXX/dist-packages/fsk/XXX
```

Check whether the parent folder of these paths (up to `dist-packages`) is present in `$PYTHONPATH`:

Check Command

```
echo $PYTHONPATH
```

If it is missing, append it to your shell configuration (adapt the path to your system):

Configuration Command

```
echo "export PYTHONPATH='/usr/local/XXX/dist-packages/:$PYTHONPATH'"
\
                            » ~/.bashrc
```

Notes

After a successful installation, the module is available to GNU Radio. If you later modify Python sources in `gr-fsk`, you can propagate changes with:

Update Commands

```
cd build/
sudo make install
```

F.1.3 Obtain and configure the evaluation flowgraph

Download the provided GNU Radio Companion flowgraph `eval_setup.grc` from the GitHub repository: https://github.com/Qwaks913/SDR_DE10_AT86RF215 to a working directory on the host computer.

GitHub Repository

The GNU Radio flowgraph is available at: https://github.com/Qwaks913/SDR_DE10_AT86RF215

Open it in GNU Radio Companion and locate the embedded Python block `rf215_cfg_sender.py`. Edit the line that defines the destination of the configuration packets so it matches your DE10-Nano HPS IP address (and adjust the port if needed):

Configuration in rf215_cfg_sender.py

```
IP, PORT = "10.42.0.78", 6000
```

Use the actual HPS address assigned on your network (e.g. via DHCP or static config) and, if the daemon listens on a different port, substitute that value. Save the block, then generate (Build) the flowgraph so the change is applied before running it.

F.2 Software installation on the HPS

GitHub Repository

The complete source code for the HPS software (including the control daemon and C programs) is available in the `final_c_code_2` directory at: https://github.com/Qwaks913/SDR_DE10_AT86RF215

F.2.1 Installing the control daemon

Copy the provided folder `final_c_code_2` anywhere on the machine, edit the `config_daemon_transceiver.service` to point to that path, then enable and start the service. The Python daemon is `config_daemon.py`.

Edit these two lines in the unit file

Service File Configuration

```
WorkingDirectory=/absolute/path/to/final_c_code_2
ExecStart=/usr/bin/python3 \
          /absolute/path/to/final_c_code_2/config_daemon.py
```

Commands to install and activate the service

Installation Commands

```
cp -r final_c_code_2 /root/final_c_code_2
chown -R root:root /root/final_c_code_2

cp /root/final_c_code_2/config_daemon_transceiver.service \
   /etc/systemd/system/config_daemon_transceiver.service

systemctl daemon-reload
systemctl enable config_daemon_transceiver.service
systemctl start config_daemon_transceiver.service
```

F.2.2 Compiling the program

Adjust default network destination (if needed). By default the executable will send UDP frames to the PC address and port hard-coded in `dma.c`. Edit the following lines before compiling if your host uses a different interface or port:

Configuration in dma.c

```
#define DEFAULT_UDP_IP   "10.42.0.1"   // default PC address
#define DEFAULT_UDP_PORT 5000           // default UDP port
```

Set them to the target host IP (and port, if required).

In the `final_c_code_2` folder, run:

Compile Commands

```
make clean && make
```

This compiles the C program and produces the executable `example_spi_dma`.

F.2.3 Running the program

The program can be run manually, but it is intended to be started by the daemon. To run it manually, use:

Manual Execution

```
./example_spi_dma [RUNNING_MODE] [IP_ADDRESS] [PORT]
```

The option `RUNNING_MODE` can be either `UDP` or `TEST`. The first mode is for normal operation, while the second is for testing purposes and will only dump the received data to the console and possibly save it to a file. The `IP_ADDRESS` and `PORT` parameters are used to specify the destination for the UDP packets. These parameters are not mandatory, as the default IP address and port are already set in the code. If you want to use the default values, you can simply run:

Example Usage

```
./example_spi_dma UDP
```

Parameters loading The program expects the configuration parameters to be sent by the daemon. When a configuration packet is received, the daemon will write it in the `config.txt` file. The parameters can also be manually edited in this file if needed.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**